
PHAR LAP
386|LINK
REFERENCE
MANUAL
386|DOS-EXTENDER SDK

# 386 | LINK Reference Manual

**Phar Lap Software, Inc.**
60 Aberdeen Avenue, Cambridge, MA 02138
(617) 661-1510, FAX (617) 876-2972
dox@pharlap.com
tech-support@pharlap.com

First Edition: October 1986
Second Edition: June 1988
Third Edition: January 1991

# Table of Contents

## Figures and Tables

# Preface

This manual, *386 | LINK Reference Manual*, contains information for the programmer who is already familiar with developing applications for the Intel 80*x*86 family of microprocessors. As a member of the Phar Lap 386 | DOS-Extender Software Development Kit (SDK), 386 | LINK links modules from a wide variety of compilers and assemblers, including products from MetaWare, Watcom, Zortech, SVS, Microway, Alsys, and Telesoft, and Phar Lap's 386 | ASM assembler.

386 | LINK joins modules for 32-bit protected mode programs, which can execute under the Phar Lap 386 | DOS-Extender, as well as for 16-bit real mode applications, which can run under MS-DOS. For 32-bit protected mode programs, 386 | LINK accepts object modules in the Easy OMF-386 format (defined in Appendix C). For 16-bit real-mode MS-DOS programs, 386 | LINK accepts the industry-standard OMF-86 object format.

In this manual are six chapters, which contain: introductory information in chapter 1, the basic usage of 386 | LINK in chapter 2, command line switches in chapter 3, switches that provide symbols to debuggers in chapter 4, switches that control and define the map file in chapter 5, and segment ordering and library searching rules in chapter 6. There are also four appendices: a summary of switches; details on error messages, on object module formats, and on MS-DOS Executable File Formats.

## Related Documentation

The 386 | DOS-Extender SDK includes these documents:

*386 | DOS-Extender Reference Manual*
*386 | DEBUG Reference Manual*
*386 | ASM Reference Manual* and
the utility guides for: CFIG386, 386 | LIB and BIND386
and related reference cards.

If you find that you have comments or suggestions for improvements, please call us, write us or send us mail at:

Phar Lap Software, Inc.
60 Aberdeen Avenue, Cambridge, MA. 02138
(617) 661-1510, FAX (617) 876-2972
dox@pharlap.com
tech-support@pharlap.com

This manual was printed on recycled, acid-free paper.

*This manual was produced on a Macintosh IIcx, using MS-Word and MacDraw. The examples in this manual were developed with 386|LINK, version 3.0.*

Chapter 1

# Introduction

386 | LINK is the Phar Lap linker for Intel 8086 and 80386 microprocessors. A member of the 80386 Software Development Series (SDK), 386 | LINK combines object modules produced by an assembler and/or a compiler into a single application program.

32-bit protected mode programs linked by 386 | LINK can execute under the Phar Lap 386 | DOS-Extender. Real mode applications can be executed under MS-DOS.

For 32-bit protected mode program, 386 | LINK accepts object modules in the Easy OMF-386 format defined in Appendix C. This format is output by most 80386 compilers, including products from MetaWare, Watcom, Zortech, SVS, Microway, Alsys, and Telesoft, and Phar Lap's 386 | ASM assembler.

For 16-bit real-mode MS-DOS program, 386 | LINK accepts the industry-standard OMF-86 object format. This format is produced by all language tools for building MS-DOS programs, including the family of tools from Microsoft and Borland.

Phar Lap Software also offers an embedded systems linker, LinkLoc, which supports a number of additional input and output object formats, automatic building of systems tables (GDT, LDT, TSS), and powerful segment and program control switches. LinkLoc is also available as a cross-development tool on VAX/VMS and UNIX systems.

Chapter 2

# Using 386 | LINK

This chapter describes the basic use of 386 | LINK including:

- the command line syntax for both versions of 386|LINK: protected and real mode;
- using indirect command files and conditional command files for repeating links without having to re-enter the switches
- using default switch settings of many 386|LINK switches
- selecting object files: 16-bit real mode files that conform to the standard Intel/Microsoft OMF-86 format and 32-bit protected mode files that conform to the Phar Lap Easy OMF-386 format.
- specifying library files
- automatically binding programs with the Phar Lap 386|DOS-Extender.

For more information on library files and the 386 | LINK switch, -LIB, please see Chapter 3 of this manual.

## 2.1 Command Line Syntax

The 386 | LINK distribution disks include two linker versions, 386LINK.EXE and 386LINKR.EXE. 386LINK.EXE is a protected-mode version that only runs on 80386- or 80486-based PCs; it runs faster and can link larger programs than 386 | LINKR.EXE. 386LINKR.EXE is a standard real-mode MS-DOS program. Always use 386LINK.EXE unless you are using a 80286- or 8086-based PC for program development.

The command line to run 386 | LINK is the linker name, followed by a list of object files, with optional command switches. The command line is:

```
386LINK filename(s) switch(es)
```

**Example:**

```
386LINK hello utils -lib clib -nomap
```

In the example,"hello" and "utils" are the names of the two object files. The example also uses two command line switches (-lib and -nomap). The -lib switch specifies the name of an object library (clib) to be linked with the program; the -nomap switch instructs 386|LINK not to produce a map file.

386|LINK supports indirect command files (please see section 2.2). The command line remains the same with the exception of preceding the filename with an "@" (at sign) character. For example:

```
386LINK @cadsoft
```

386|LINK switches can optionally be placed in an environment variable named 386LINK. For example:

```
set 386link=-lib clib -nomap
386link hello utils
```

To display the 386|LINK version number and a summary of valid command line operands, type the name of 386|LINK without any command line arguments. The summary specifies the name of each switch along with any required arguments, followed by a few words explaining the switch function. For example:

```
386|LINK: 3.0 -- Copyright (C) 1986-91 Phar Lap Software, Inc.

386LINK  file1,file2,...          Input files [.OBJ]
         -LIB file1,file2,...     Library files [.LIB]
         -EXE file                .EXE output file [.EXE]
         -RELEXE file             Reloc .EXE output file
         -NOOUTPUT                No output file
         -MAP file                Map file [.MAP]
         -NOMAP                   No map file
```

### 2.1.1 File Name Syntax

Command line file names can be specified as a complete file name (*filename.extension*) or can be given without an extension, in which case 386 | LINK supplies a default extension. The table below lists the default filename extensions assumed by 386 | LINK.

| File Type | Default Extension |
|---|---|
| Object file | .OBJ |
| Library file | .LIB |
| Command file | .LNK |
| Map file | .MAP |
| Real-mode executable | .EXE |
| Protected-mode executable | .EXP |
| Protected-mode relocatable executable | .REX |

A complete or partial path may be specified with the file name. The file path must follow the standard MS-DOS syntax of file paths. If no file path is given, then the current default device and directory are assumed.

### 2.1.2 Command Switch Syntax

Command switches begin with the character "-" (minus sign), followed by the name of the switch. No spaces are allowed between the minus sign and the switch name. Switch arguments, if any, must immediately follow the switch name, separated by spaces. Many 386 | LINK switches have both a short form and a long form and they can be used interchangeably.

Input file names and switches can be placed in any order on the command line. Adjacent input file and module names can be separated by spaces and/or commas. However, adjacent switches, or an adjacent switch and input file name, must be separated by spaces. Input file names can not begin with a minus sign character, so that 386 | LINK can distinguish between file names and switches.

If conflicting switches are specified on the command line, the last (rightmost) switch takes precedence.

### 2.1.3 Numbers Used with 386 | LINK Switches

Many 386 | LINK switches require a number as an argument. By default, all numbers are assumed to be decimal numbers. A different radix can be specified by appending a radix specifier to the number. Please see the following table for a list of the bases and their radix specifiers.

| Base | Radix Specifier |
|---|---|
| 2 | B |
| 8 | Q |
| 10 | None |
| 16 | H |

The underscore character ("_") can be used in numbers to make them more readable; for example, 80000000h is the same as 8000_0000h.

**Example:**

The following examples all specify the same value for the -MAXDATA switch:

```
-maxdata 4096
-maxdata 1000h
-maxdata 1_0000q
```

## 2.2 Indirect Command Files

If you have too many 386 | LINK switches and files to fit on the command line, place them in an indirect command file. Enter 386 | LINK command line parameters in an indirect file in the exactly the same manner entered as on the command line. Command files can have as many lines as needed.

For 386 | LINK to process an indirect command file, preface the name of the file with an @ character. When 386 | LINK encounters a command line parameter which begins with an @, it opens the file and processes the

commands in it. If the file name does not have an extension specified, a default of ".LNK" is assumed. For example, the command 386LINK @mylink causes 386 | LINK to read its command string from the file "MYLINK.LNK."

386 | LINK also supports conditional command files. To invoke a conditional command file, type the @ character twice before the filename, e.g., @@filename. If the specified conditional command file is not present, 386 | LINK continues without signalling an error.

Indirect command files can be used both with other command line switches and other indirect command files. Multiple indirect files are processed left to right in the order they are encountered on the command line.

**Example:**

```
386LINK @mylink -symbols
386LINK @exec @bios @bootldr
```

When the command line scanner generates an error within a command file, it prints out the command file name and the line number within the file as part of the error message.

Comments in command files are specified with the exclamation point character (!). A comment can begin anywhere on a line, and all text after it is ignored.

The following example indirect command file uses comments for clarity:

```
! Linking a C program for the 8086
!
init            ! C initializer
main            ! The program to be linked
-lib clib       ! The C run-time library
-exe exam       ! Name of the .EXE file
-8086           ! Target CPU is 8086
```

## 2.3 Default Switch Settings

Command line switches override 386 | LINK defaults. By default, 386 | LINK makes the following assumptions:

- target CPU is an 80386 (-80386)
- 386|LINK output file type is Phar Lap .EXP (-EXE)
- no object libraries to be searched (-LIB)
- produce a map file (-MAP)
- starting offset of the program segment is zero (-OFFSET 0)
- ignore case of symbols (-ONECASE)
- do not produce a symbol table in the output file (-NOSYMBOLS)
- size of the stack segment is zero (-STACK 0)
- public symbols are sorted alphabetically in the map file (-PUBLIST BYNAME)
- all available data is allocated to the program (-MINDATA 0, -MAXDATA 0FFFFFFFFh)
- all 386|DOS-Extender switches are given default values (-MINREAL 0, -MAXREAL 0, -MINIBUF 1, -MAXIBUF 16, -NISTACK 6, -ISTKSIZE 1, -REALBREAK 0, -CALLBUFS 0, -PRIVILEGED)
- object records do not require checksums (-NOCHECKSUM).

## 2.4 Object Files

386 | LINK operates by reading one or more relocatable object files and producing a single executable program file. These object files can be produced by the Phar Lap 386 | ASM assembler or any compatible high-level language compiler.

386 | LINK can process two different types of object files:

- 16-bit real mode object files
- 32-bit protected mode object files.

A 16-bit real mode object file must follow the standard Intel/Microsoft OMF-86 file format definition. The majority of currently available MS-DOS compilers produce this OMF-86 format. A 32-bit protected mode object file must follow the Phar Lap Easy OMF-386 format as documented in Appendix C. This format is output by most 80386 compilers, including products from MetaWare, Watcom, Zortech, SVS, Microway, Alsys, Telesoft, and LPI.

The names of the object files appear immediately after 386 | LINK on the command line. File names can be separated by either spaces or commas. If no file name extension is specified, then 386 | LINK assumes a default extension. If no directory path is specified, then the current directory is assumed. These are examples of command lines with object files in them:

```
386LINK hello
386LINK main,sub1.o,sub2.o,sub3.o
386LINK main \lib\init
```

386 | LINK processes object files in the order they appear on the command line. The only instance when the order is important is when an object module that sets up segment ordering for an application is being used. This type of object module should appear first on the command line. Please see Chapter 6 for more information on segment ordering.

## 2.5 Library Files

Most high-level language compilers include libraries of useful functions which can be used by an application under development. Typical functions found in run-time libraries include:

- program initialization code
- file I/O functions
- floating point arithmetic functions
- string manipulation functions
- formatting functions.

These functions, which may number in the hundreds, are collected together into large files called libraries. These libraries are then linked with the application after it is compiled. Only those parts of library files specifically referenced by the application or other parts of the library are included in the program. This reduces the size of the final executable image, because it does not contain any unneeded code.

Library files are identified with the -LIB switch. Please see section 3.1.

## 2.6 Autobinding the 386 | DOS-Extender Stub Loader

386 | LINK has the ability to bind the stub loader program, STUB386.EXE, to the front of an application .EXP file. The resulting .EXE file can be run by typing the file name, just like a real mode DOS program. The stub loader program searches the execution PATH for RUN386.EXE (the 386 | DOS-Extender executable) and loads it; 386 | DOS-Extender then loads the application .EXP file following the stub loader in the bound .EXE file.

To autobind STUB386.EXE to an application .EXP file and create a bound executable, specify STUB386.EXE as one of the input object files on the command line. For example, to create the bound file HELLO.EXE:

```
386link hello stub386.exe -lib clib
```

## 2.7 Autobinding 386 | DOS-Extender

For customers who have purchased a 386 | DOS-Extender run-time license, 386 | LINK has the ability to bind 386 | DOS-Extender automatically to the program .EXP file. This avoids the need to run the BIND386 utility program as a separate step.

There are two ways to autobind 386 | DOS-Extender:

- Specify the bindable 386|DOS-Extender (RUN386B.EXE) as one of the input object files on the command line.

- Add RUN386B.EXE to a library. One of the other object modules in the link must explicitly reference the public symbol $$RUN386B ($$ followed by the module name) assigned to .EXE files in libraries. This is most easily done with an EXTRN statement in an assembly language file, e.g., EXTRN $$RUN386:BYTE.

The following three methods of linking and binding HELLO.EXE are equivalent:

- 386link hello -lib clib
  bind386 run386 hello

- 386link hello run386b.exe -lib clib

- assuming one of the modules in HELLO.EXP references $$RUN386B:

  386lib clib -add run386.exe
  386link hello -lib clib

The same techniques can be used to autobind the run-time version of 386|VMM. The following are equivalent:

- 386link hello -lib clib
  bind386 run386 vmmdrvb hello

- 386link hello run386b.exe vmmdrv.exp -lib clib

- assuming one of the modules in HELLO.EXP references $$RUN386B and $$VMMDRVB:

  386lib clib -add run386.exe vmmdrvb.exp
  386link hello -lib clib

Chapter 3

# 386 | LINK Switches

This chapter describes the 386 | LINK switches and how to use them. Within the description of each switch, the long and short form is included as well as an actual example. The switches are presented according to function, appearing in alphabetic order within the divisions listed below:

- library files switches
- target cpu switches
- output format switches
- suppressing the output file
- program parameter switches, which affect the initial program runtime state, including some 386|DOS-Extender run-time switches
- map file switches, including local and public symbol information as well as dumping object files into the map file
- symbol table switches, including a Microsoft CodeView format symbol table.
- 386|DOS-Extender switches
- linker operation and control
- register values in Intel Symbol table switch
- treating local symbols as data switch.

In addition there are helpful sections on managing aspects of linking such as controlling the map file and managing public symbols. For information on basic usage of 386 | LINK such as the command line syntax, please see Chapter 2 of this manual.

## 3.1 The -LIB Switch

The "-LIB" switch specifies one or more library files. The names of the library files immediately follow the switch, separated by either spaces or commas. If no file name extension is specified for a library file, then 386 | LINK assumes the extension, .LIB. Please see Chapter 2 for details.

The -LIB switch may be used multiple times in a single 386 | LINK command string. 386 | LINK builds a list of the library files and processes them in the order they were specified on the command line. For information on the 386 | LINK search algorithm please see section 6.3.

If a library file is specified as an object input file (the -LIB switch is not used), then 386 | LINK includes all of the object modules in the library, rather than only those referenced by other parts of the program.

To create and maintain libraries of commonly used functions use the Phar Lap library utility program, 386 | LIB. Consult the *386 | LIB Utility Guide* for details.

**Syntax:**

```
-lib file1 ... filen
```

**Short Form:**

```
-l file1 ... filen
```

**Example:**

```
386link hello -lib clib
386link main -lib \lib\clib \lib\math \lib\graphics
```

## 3.2 Target CPU Switches

The target CPU switches indicate the type of CPU on which the program will execute. For CPUs which support both real and protected mode, the target CPU switch also specifies the processor mode used to execute the program.

### 3.2.1 The -8086 Switch

The -8086 switch tells 386 | LINK that the program being linked will execute on an 8086/186 CPU or, in real mode, on an 80286/80386 CPU. In real mode, segment selectors specify physical addresses which appear on 16-byte boundaries. Any segment in the load image cannot exceed 64K bytes in size, and a 20-bit address size limitation (1Mb) is enforced in the final load image.

Programs which execute in real mode on 80286 or 80386 CPUs can take full advantage of any additional instructions and address modes specific to that CPU, though they are still limited by the 1Mb address space limitation. This feature is particularly useful for 80386 specific programs, because they can take advantage of the 32-bit wide registers for mathematical and block transfer operations. When assembling code for the 386 which will execute in real mode, the -386R switch should be used with 386 | ASM. This switch causes the assembler to generate a 16-bit real mode object file, with any 32-bit instructions preceded by the necessary override bytes. It is also possible to assemble real mode code with the
-80386 assembler switch, as long as all the segments in the program are USE16. Please see the *386 | ASM Reference Manual* for more details.

The output format used for real-mode programs is the standard MS-DOS .EXE file format (please see Appendix D).

**Syntax:**

```
-8086
```

**Short Form:**

```
-86
```

**Example:**

```
386link hello -8086
386link mathlib sin cos -86
```

### 3.2.2 The -80386 and -80486 Switches

The -80386 and -80486 switches are synonyms. They tell 386 | LINK that the program being linked will run on an 80386 or 80486 CPU in 32-bit protected

mode. For 32-bit protected mode, 386 | LINK enforces a 32-bit segment size limit (4 GB) and a 32-bit address size limit (4 GB). 386 | LINK support for 32-bit protected mode programs is tailored to the needs of applications which run under the Phar Lap 386 | DOS-Extender.

The default output format for 32-bit protected programs is Phar Lap's .EXP format. The -RELEXE switch can be used to select the relocatable .REX file format. Please see Appendix D for executable file formats.

**Syntax:**

```
-80386
-80486
```

**Short Form:**

```
-386
-486
```

**Example:**

```
386link hello386 -80386
386link cad386 -lib hcc -486
```

## 3.3 Output Format Switches

The output format switches instruct 386 | LINK which format to use for the output file after the link process is complete. 386 | LINK can write only one output file per image linked, so only one output format switch can be specified at a time. The choice of output format depends upon both the target CPU and output format switches. This interaction is summarized in the table below.

| | -8086 | -80386 |
|---|---|---|
| -EXE | MS-DOS .EXE | Phar Lap .EXP |
| -RELEXE | Invalid | Relocatable .REX |

If no output format is specified, the default format is:

- the Phar Lap .EXP format for protected mode (80386) targets
- the MS-DOS .EXE format for an 8086 target.

The default output file name is the name of the first object file with an extension of either .EXP (for protected-mode programs) or .EXE. (for real-mode programs) The output file is not automatically placed in the current directory; by default, it is placed in the same directory as the first object file on the command line.

For information on how to change both the output format and output file name by using the output format switches, please see the following sections.

### 3.3.1 The -EXE Switch

The -EXE switch instructs 386 | LINK to produce either a Phar Lap .EXP file for an 80386 target or a MS-DOS .EXE file for an 8086 target. The Phar Lap .EXP format is the format loaded by 386 | DOS-Extender.

The switch takes a single argument, which is the path and name of the output file. If no extension is specified, the default is assumed. If no path information is specified with the file name, then the output file is placed in the current directory.

More information on both the MS-DOS .EXE and the Phar Lap .EXP file formats appears in Appendix D.

**Syntax:**

```
-exe file
```

**Short Form:**

```
-e file
```

**Example:**

```
386link hello -exe d:\exelib\hello.exe
386link main -exe maintest
```

### 3.3.2 The -RELEXE Switch

The -RELEXE switch instructs 386 | LINK to produce a relocatable 80386 protected mode file. This switch is valid only for an 80386 target. This switch takes a single argument, which is the path and name of the output file. If no extension is specified, the default, .REX, is assumed. If no path information is specified with the file name, then the output file is placed in the current directory.

The .REX format allows for relocation of a program by a loader at execution time. Phar Lap's 386 | DOS-Extender does not use .REX format. More information on the format of a .REX load image appears in Appendix D.

**Syntax:**

```
-relexe file
```

**Short Form:**

```
-relexe file
```

**Example:**

```
386link hello -relexe d:\exelib\hello.exe
386link main -relexe maintest
```

### 3.3.3 The -NOOUTPUT Switch

The -NOOUTPUT switch causes 386 | LINK not to produce an output file. This switch can be used during the initial stages of program development, when the only purpose of the link is to check for errors.

**Syntax:**

```
-nooutput
```

**Short Form:**

```
-noo
```

**Example:**

```
386link hello -nooutput
```

### 3.3.4 Packing an .EXP File

386 | LINK can pack (perform data compression on) an .EXP file. When packing an .EXP file, 386 | LINK combines multiple instances of a single byte value (usually zero) into an encoded record. This packing scheme results in significantly smaller .EXP files, if the files contain large blocks of uninitialized or zero data. Packed files take less disk space than unpacked files. However, very large packed programs may suffer severe load-time penalties under Phar Lap's 386 | VMM virtual memory manager (please see the *386 | VMM Reference Manual*). Not all programs benefit from packing because they may not contain strings of data bytes which can be compressed. The best method of determining if a program will benefit from packing is to link the program both unpacked and packed. If the packed version is significantly smaller, then switch to using packed files.

To cause 386 | LINK to pack its .EXP output file, specify the -PACK switch on the command line. The switch takes no arguments. The -PACK switch can be specified only if the type of 386 | LINK output file is .EXP (-80386 target CPU and -EXE output format). The other output file formats cannot be packed.

By default, 386 | LINK does not produce packed .EXP files. The -NOPACK switch is, however, included for completeness.

**Syntax:**

```
-pack
-nopack
```

**Short Form:**

```
-pack
-nopack
```

**Example:**

```
386link bigfft -lib \fortran\flib -pack
```

## 3.4 Program Parameter Switches

The program parameter switches affect the initial program runtime state. These parameters control the amount of extra data to allocate to the program, the size of the stack in the program, the base offset at which the program is linked, and selected 386 | DOS-EXTENDER run-time switches.

### 3.4.1 The -START Switch

The -START switch specifies the entry point for a program. This switch overrides an entry point address supplied by an object module. The switch argument is the name of a public symbol in the program.

**Syntax:**

```
-start symbolname
```

**Short Form:**

```
-start symbolname
```

**Example:**

```
386link hello -start main
```

### 3.4.2 The -MINDATA and -MAXDATA Switches

When 386 | DOS-Extender loads an .EXP file, or when MS-DOS loads an .EXE file, it must allocate a block of memory large enough to hold the entire program. Optionally, additional memory beyond the end of the program can be allocated. The -MINDATA and -MAXDATA switches configure an .EXP, .EXE, or .REX file to identify how much extra memory the program needs.

The -MINDATA switch specifies the minimum number of extra data bytes which must be allocated when the program is loaded. The byte count is given after the -MINDATA switch as a numeric constant.

The -MAXDATA switch specifies the maximum number of extra bytes to be allocated when the program is loaded. Like -MINDATA, the maximum byte count is given after the switch.

When 386 | DOS-Extender loads an .EXP file, or when MS-DOS loads an .EXE file, it guarantees that at least -MINDATA bytes are available past the end of the program and makes available any additional memory, up to a maximum of -MAXDATA.

If the -MINDATA switch is not specified on the command line, then 386 | LINK assumes a default of zero bytes.

If the -MAXDATA switch is not specified on the command line, then 386 | LINK assumes a default of FFFFFFFFh for .EXP and .REX files, and FFFFFh for .EXE files, which allocates all available extra memory to the program at load time.

**Syntax:**

```
-mindata nbytes
-maxdata nbytes
```

**Short Form:**

```
-mind nbytes
-maxd nbytes
```

**Example:**

```
386link special -mindata 1000h -maxdata A000h
```

### 3.4.3 The -OFFSET Switch

When linking a program for 80386 protected mode, 386 | LINK builds the program for flat model by placing all code and data into a single segment. By default, 386 | LINK places the first byte of code or data at offset 0 in the segment. The -OFFSET switch overrides this default, starting the code and data at a different offset. The -OFFSET switch can only be used with .EXP format files.

In an application that runs under 386 | DOS-Extender, it is possible to detect null pointer references by linking the program at an offset which is a multiple of 4K bytes. At load time, 386 | DOS-Extender does not map the pages which would be unused, because they appear in the area skipped by the -OFFSET switch. If a reference is made (through a null pointer) to any location in this

unmapped area, a page fault is generated. This prevents a null pointer reference from corrupting the program's code or data segment. The offset must be an exact multiple of 4K bytes, or
386 | DOS-Extender will refuse to run the program.

The -OFFSET switch takes a single argument, which is a numeric constant specifying the starting offset of the program.

**Syntax:**

```
-offset nbytes
```

**Short Form:**

```
-off nbytes
```

**Example:**

```
386link testprog -offset 1000h
```

### 3.4.4 The -STACK Switch

The -STACK switch specifies the size of the stack area for a program. The switch must be followed by a numeric constant which specifies the number of bytes to be allocated to the stack.

If a stack segment is already present in the program, then the -STACK switch changes the size of the existing segment. 386 | LINK, however, will only increase the size of the existing stack area. If an attempt is made to decrease the size of the stack area, 386 | LINK flags an error.

If no stack segment is present in the program, then 386 | LINK creates one and places it after the last segment of the program. The name of the segment is "STACK", and it has the size specified by the switch argument.

If no stack segment is defined in the program and the -STACK switch is not specified on the command line, then 386 | LINK posts a warning message to the effect that there is no stack segment present in the program. In general, a program will not run properly without a stack segment.

**Syntax:**

```
-stack nbytes
```

**Short Form:**

```
-s nbytes
```

**Example:**

```
386link hello -stack 8192
```

### 3.4.5 386 | DOS-Extender Switches

Many of the run-time parameters for a program that executes under 386 | DOS-Extender can be specified with command switches at link time. 386 | LINK records the 386 | DOS-Extender switch values in the header of the output file, where they are processed every time the program is run. The advantage is that it is not necessary to specify parameters every time the program is run.

The following is a summary of the 386 | DOS-Extender switches which can be specified at link time:

| | |
|---|---|
| -UNPRIVILEGED | Run the program at privilege level one, two, or three. |
| -PRIVILEGED | Run the program at privilege level zero. |
| -MINREAL *nparagraphs* | Minimum number of 16-byte paragraphs of real mode memory to leave free. |
| -MAXREAL *nparagraphs* | Maximum number of 16-byte paragraphs of real mode memory to leave free. |
| -REALBREAK *symname* | Mark all bytes from the start of the program up to, but not including, the specified public symbol to be loaded in conventional memory. |
| -CALLBUFS *nkilobytes* | The amount of memory to be allocated to the intermode call buffer available to the application program for passing data on procedure calls between protected and real modes. |

| | |
|---|---|
| -MINIBUF *nkilobytes* | Minimum number of 1 KB blocks to allocate for data buffer used on DOS system calls. |
| -MAXIBUF *nkilobytes* | Maximum number of 1 KB blocks to allocate for data buffer used on DOS system calls. |
| -NISTACK *n* | The number of interrupt stack buffers to allocate; this number controls the maximum number of nested switches between protected and real modes that may be performed. |
| -ISTKSIZE *nkilobytes* | The size, in 1 KB blocks, of each interrupt stack buffer. |

For a more detailed description of these switches, see the *386 | DOS-Extender Reference Manual*.

## 3.5 Map File Switches

The 386 | LINK map file is a text file describing the output load image. It contains the following information:

- the command switches specified when the program was linked
- the names of the input object files
- a list of the segments comprising the program
- a list of the public symbols in the program
- optionally, a list of the local symbols in the program.

The primary purpose of the map file is to locate segments and symbols when debugging the program. The map file also provides a record of how a program is linked.

By default, 386 | LINK produces a map file each time a program is linked. The characteristics of the map file, as well as whether the file is actually produced, can be controlled through the use of the map file switches. The rest of this section details the map file switches and their effect. A discussion of how to interpret the map file appears in Chapter 5.

### 3.5.1 The -MAP Switch

By default, 386 | LINK produces a map file each time a program is linked. The default name of the map file is the name of the executable file, with its extension changed to .MAP. Any path information specifying a directory where the executable file is to be placed also applies to the map file. If 386 | LINK has been instructed not to generate an output file by using the -NOOUTPUT switch, a map file is still produced. In this case, the name of the map file is the name of the first object file with its extension again changed to ".MAP." Any path information is also duplicated.

The -MAP switch renames or relocates the map file. The switch takes a single argument, which is the name of the map file to be produced. If no file name extension is specified, then a default of .MAP is assumed. If no path information is specified in the map file name, then it is placed in the current directory.

**Syntax:**

```
-map file
```

**Short Form:**

```
-m file
```

**Example:**

```
386link hello -map a:hello
386link moe larry curly -map stooges
```

### 3.5.2 The -NOMAP Switch

If a map file is not needed, 386 | LINK can be prevented from producing one by using the -NOMAP switch. The switch takes no arguments.

The -NOMAP switch is useful to make 386 | LINK run faster, since no time is spent writing the map file. The switch is also a good way to save disk space, because map files can become quite large.

**Syntax:**

```
-nomap
```

**Short Form:**

```
-nom
```

**Example:**

```
386link hello -nomap
```

### 3.5.3 Displaying 386 | LINK Switches in the Map File

The -SWITCHES switch lists the command line switches which appear on the 386 | LINK command line or in any indirect command files. This is the default operation of 386 | LINK, so it is not usually necessary to specify this switch on the command line.

The -NOSWITCHES switch causes 386 | LINK not to list the command line switches on 386 | LINK command line or in any indirect command files. This switch is available to provide map file compatibility with earlier versions of 386 | LINK. It also saves a small amount of space in the map file.

**Syntax:**

```
-switches
-noswitches
```

**Short Form:**

```
-sw
-nosw
```

**Example:**

```
386link hello -nosw
```

### 3.5.4 Map File Formatting Switches

The -MAPNAMES switch controls the length of global symbol names displayed in the map file. By default, segment, group, class, module, and public symbol names are truncated to 12 characters in the map file. The switch takes a numeric constant argument which increases the length of global symbols in the map file to the specified number of characters. Increasing the symbol name length may cause the default maximum line

width of 80 characters to be exceeded. If this occurs, 386|LINK prints less information about segments and public symbols. This loss of information can be prevented by using the -MAPWIDTH switch.

The -MAPWIDTH switch controls the maximum line width in the program map file. The switch takes a numeric constant argument which is the new maximum width for lines in the map file. If this switch is not used, a default of 80 characters per line is used.

**Syntax:**

```
-mapnames nchars
-mapwidth nchars
```

**Short Form:**

```
-mapn nchars
-mapw nchars
```

**Example:**

```
386link hello -mapn 30 -mapw 120
```

### 3.5.5 The -FULLSEG Switch

The -FULLSEG switch breaks down each segment which makes up the load image into its constituent parts from the object files. The breakdown shows the load address and size of each segment piece and what object module the piece comes from.

**Syntax:**

```
-fullseg
```

**Short Form:**

```
-fullseg
```

**Example:**

```
386link hello -fullseg
```

### 3.5.6 Public Symbol Ordering Switch

The -PUBLIST switch controls the ordering of the list of public symbols. By default, 386 | LINK lists public symbols alphabetically.

The -PUBLIST BYVALUE switch sorts the list of public symbols in the program by value. This option is useful when using the map file to find out what routine or variable resides at a particular memory location.

The -PUBLIST BYNAME switch sorts the list of public symbols which make up the program alphabetically. This is the default operation of 386 | LINK.

The -PUBLIST BOTH switch produces two listings of the public symbols: one sorted alphabetically and one sorted by value.

The -PUBLIST NONE switch causes 386 | LINK not to list the public symbols which make up the program at all. This option is useful for reducing the size of the map file.

**Syntax:**

```
-publist byname
-publist byvalue
-publist both
-publist none
```

**Short Form:**

```
-publ byname
-publ byvalue
-publ both
-publ none
```

**Example:**

```
386link hello -publ byvalue
386link hello -publist both
```

### 3.5.7 Local Symbol Information

The -LOCMAP and -NOLOCMAP switches select whether local symbol information is placed in the map file. Applications developers should not select this option, which makes the map file very large. The primary purpose of listing local symbol application is checking for correct operation of programs, such as source code debuggers, which process local symbol information in the symbol table.

The -LOCMAP switch displays full symbolic information in the map file. This includes any typedef, line number, and local symbol information that appear in both Intel and Phar Lap object modules.

For object input files that contain CodeView style symbolic information, the -LOCMAP switch writes this information in the map file in decoded, English language format. For object input files that contain Intel style symbolic information, the -LOCMAP switch writes this information in hexadecimal form.

The -NOLOCMAP switch suppresses the display of source code symbolic information in the map file. This includes all typedef, line number, and local symbol information which appear in both Intel and Phar Lap object modules. This switch is the default operation of 386|LINK.

These switches take no arguments.

**Syntax:**

```
-locmap
-nolocmap
```

**Short Form:**

```
-locm
-nolocm
```

**Example:**

```
386|LINK hello -locmap
386|LINK hello -nolocm
```

### 3.5.8 Dumping Object Files in the Map File

The -DUMP switch dumps the hexadecimal object records for each object file processed at the head of the map file. This option is useful for finding the exact location of an error in an illegal .OBJ file. The switch takes no arguments and only affects the 386|LINK map output file. It does not affect the linking process in any way.

The object files are in the front of the map file immediately after the command switch listing. Each record is displayed in hexadecimal, along with its hexadecimal offset in the file. A typical listing looks like this:

```
Offset in file 0x00000000
0000:  80 07 00 05 48 45 4C 4C 4F 00

Offset in file 0x0000000A
0000:  88 08 00 80 AA 38 30 33 38 36 3D

Offset in file 0x00000015
0000:  88 14 00 80 00 48 69 67 68 20 43 20 31 2E 33 2E
0010:  20 53 4D 41 4C 4C 88
```

**Syntax:**

```
-dump
```

**Short Form:**

```
-dump
```

**Example:**

```
386link badfile -dump
```

## 3.6 Symbol Table Switches

By default, 386|LINK does not write a symbol table in the executable file. When debugging a program, it is usually desirable to include a symbol table in the executable file so the debugger can support symbolic references to program locations and data. The sections below describe 386|LINK switches that control symbol table generation. Please see also Chapter 4.

### 3.6.1 Selecting a Symbol Table

386 | LINK can generate four different symbol table formats in the executable file:

- Phar Lap PubSym format: this format includes only public symbols and is used with debuggers, such as Phar Lap's 386|DEBUG, which do not support source code debugging.
- Phar Lap FullSym format: this format includes full local symbol, typedef, and line number information in addition to public symbols. It is used with source code debuggers such as Phar Lap's 386|SRCBug.
- Microsoft CodeView symbol table format: this format includes full source code debugging information, and is used with debuggers, such as MetaWare's mdb debugger, that read CodeView style symbol tables.
- Intel OMF-386 symbol table format: this format includes full source code debugging information and is used with debuggers that read Intel style symbol tables.

Phar Lap PubSym, Phar Lap FullSym, and Microsoft CodeView symbol table formats have a 16-bit and a 32-bit format. Intel OMF-386 symbol table format is 32-bit only. For 32-bit .EXP files, a 32-bit symbol table is generated, and the file header specifies the symbol table location and size. For 16-bit .EXE files, a 16-bit symbol table is generated and placed at the end of the file, following the program code and data. For 32-bit .REX files, a 32-bit symbol table is generated and placed at the end of the file, following the program code and data. Table 3-1 summarizes symbol formats available.

**TABLE 3-1**
Symbol Formats

| | Switch | | | |
|---|---|---|---|---|
| **Executable File Format** | **-SYMBOLS** | **-FULLSYM** | **-CVSYMBOLS** | **-ISYMBOLS** |
| MS-DOS 16-bit .EXE | Phar Lap PubSym | Phar Lap FullSym | Microsoft CodeView | not supported |
| Phar Lap 32-bit .EXP | Phar Lap PubSym | Phar Lap FullSym | Microsoft CodeView | Intel OMF-386 |
| Phar Lap 32-bit .REX | Phar Lap PubSym | Phar Lap FullSym | Microsoft CodeView | Intel OMF-386 |

## The -NOSYMBOLS Switch

The -NOSYMBOLS switch specifies that no symbol table is included in the executable file. This is the default operation of 386 | LINK.

**Syntax:**

```
-nosymbols
```

**Short Form:**

```
-nosym
```

**Example:**

```
386link hello -nosym
```

## The -SYMBOLS Switch

The -SYMBOLS switch places a Phar Lap PubSym format symbol table in the executable file. This symbol table format is documented in Appendix D.

**Syntax:**

```
-symbols
```

**Short Form:**

```
-sym
```

**Example:**

```
386link hello -sym
```

## The -FULLSYM Switch

The -FULLSYM switch places a Phar Lap FullSym format table in the executable file. If you are writing a debugger or other program that processes FullSym symbol tables, contact Phar Lap for documentation on the symbol table format.

**Syntax:**

```
-fullsym
```

**Short Form:**

```
-fullsym
```

**Example:**

```
386link hello -fullsym
```

## The -CVSYMBOLS Switch

The -CVSYMBOLS switch places a Microsoft CodeView format symbol table in the executable file. See section 4.4 for information on obtaining documentation of the CodeView symbol table format from Microsoft.

**Syntax:**

```
-cvsymbols
```

**Short Form:**

```
-cvsym
```

**Example:**

```
386link hello -cvsym
```

## The -ISYMBOLS Switch

The -ISYMBOLS switch places an Intel OMF-386 format symbol table in the executable file. This switch cannot be used if the executable file is a 16-bit MS-DOS .EXE file. See section 4.4 for information on obtaining documentation of the OMF-386 symbol table format from Intel.

**Syntax:**

```
-isymbols
```

**Short Form:**

```
-isym
```

**Example:**

```
386link hello -isym
```

## 3.6.2 Selecting Case Sensitivity

By default, 386 | LINK ignores the case of public symbols which make up the program being linked. For example, the symbols "abc," "ABC," and "aBc" are equivalent in 386 | LINK. Some programming languages require case differentiation between public symbol names, treating two symbols which are spelled the same but with different case as two different symbols. For compatibility with these languages and their case-sensitivity, 386 | LINK supports the -TWOCASE switch.

The -ONECASE switch ignores the case of public symbols. This is the default operation of 386 | LINK.

**Syntax:**

```
-onecase
-twocase
```

**Short Form:**

```
-oc
-tc
```

**Example:**

```
386link hello -lib \lib\clib -twocase
386link hello -lib \pascal\plib -onecase
```

## 3.6.3 Managing Public Symbols

Public symbols can be created, redefined, and exported from one link for inclusion in a second link. The sections below describe the switches used for public symbol management.

## The -SYMFILE Switch

The -SYMFILE switch creates an output file containing one entry for each public symbol in the program. The symbol file is a text file which can be used for later links as an indirect command file, allowing one program to reference symbols inside of another program. In addition, this file can also be processed by other utility programs for symbol information for debuggers and time profilers.

Often only a few symbols, rather than all public symbols in the program, are desired. Use the -EXPORT switch to limit the symbols placed in the symbol file.

For 32-bit links, the symbol offset is placed in the symbol file in the following format:

```
DEFINE MAIN=00001B8
```

For 16-bit links, the symbol segment and offset are placed in the symbol file:

```
-DEFINE INIT=005E:0259
```

The -SYMFILE switch takes a single parameter, *filename*, which is the name of the symbol file to be created. If no extension is specified for the file, a default of ".SYM" is added.

**Syntax:**

```
-symfile filename
```

**Short Form:**

```
-symf filename
```

**Example:**

```
386link hello -symfile hello
```

## The -EXPORT Switch

The -EXPORT switch selects public symbols to place in a symbol file. The -SYMFILE switch must also be used to specify the name of the symbol file. Only symbols specified with the -EXPORT switch (rather than all public symbols) are placed in the symbol file.

**Syntax:**

```
-export sym1, ... symn
```

**Short Form:**

```
-export sym1, ... symn
```

**Example:**

```
386link hello -symfile hello -export main,init,cleanup
```

## The -DEFINE Switch

The -DEFINE switch creates symbols in the public symbol table. This switch takes an argument of *symbol=value* with *symbol* as the defined symbol name and *value* as the address value. Optional segment selector values can be included.

**Syntax:**

```
-define symbol=[seg:]offset
```

**Short Form:**

```
-def symbol=[seg:]offset
```

**Example:**

```
386link hello -def buffsize=4096
```

## The -REDEFINE Switch

The -REDEFINE switch changes the value of an existing symbol in the public symbol table. This switch takes an argument of *symbol=value* to change. Arguments are the same as for the -DEFINE switch described above, except that this switch alters an existing symbol, rather than creating a new one.

**Syntax:**

```
-redefine symbol=[seg:]value
```

**Short Form:**

```
-redef symbol=[seg:]value
```

**Example:**

```
386link hello -stack 8192 -redef stksize=8192
```

## 3.6.4 The -ATTRIBUTES Switch

The -ATTRIBUTES switch sets or changes various segment characteristics or attributes in a descriptor table entry. This is useful when creating a symbol table for a protected mode program, to correctly place code symbols in the code segment (selector 000Ch), and data symbols in the data segment (selector 00014h). See section 4.3 for details.

If you get target/frame conflict errors when you use the -ATTRIBUTES switch, one or more assembly language source modules have EXTRN directives incorrectly placed. Make sure that the EXTRN statement for a public symbol is either placed inside the segment in which it is located, or outside all segment blocks in the source module.

**Syntax:**

```
                | default                |
                | segment segmentname    |     | er |
-attributes     | group groupname        |     | eo |
                | class classname        |     | rw |
                | selector number        |     | ro |
```

**where**

```
segmentname = name of segment whose attributes change
groupname = name of group whose segment attributes change
classname = name of class whose segment attributes change
number =  selector number of segment whose attributes change
eo = code segment, execute only
er = code segment, read only
ro = data segment, read only
rw = data segment, read and write
```

**Short Form:**

```
                 | def                |
                 | seg segmentname    | | er |
-attr            | gr groupname       | | eo |
                 | cl classname       | | rw |
                 | sel number         | | ro |
```

**Example:**

```
386link hello -lib clib
-attributes group cgroup er
-attr group dgroup rw -fullsym
```

## 3.6.5 The -PURGE Switch

The -PURGE switch removes symbolic information for specific modules from the symbol tables created in the executable output file. The -PURGE switch is most often used to limit the size of the symbol table in the output file so that it can be loaded more quickly by a debugger. The wildcard character (*) can be used in module names.

**Syntax:**

```
          | types   |
          | lines   |
-purge    | publics | . . modulename . . .
          | locals  |
          | all     |
          | none    |
```

**where:**

```
types = purge typedefs information
lines = purge line numbers information
publics = purge public symbols
locals = purge local symbols (automatic variables)
none = do not purge symbol information
all = purge all symbol information (typedefs, line
      numbers, publics, and locals).
```

**Short Form:**

```
          | types   |
          | lines   |
-pur      | publics | . . modulename . . .
          | locals  |
          | all     |
          | none    |
```

**Example:**

To purge all symbolic information from the startup and bigutils input modules:

```
386link hello -lib clib -purge all startup,bigutils
```

To purge typedef information from all input modules:

```
386link hello -lib clib -purge types *
```

To purge typedef and line number information from input modules starting with "kern":

```
386link hello -lib clib -purge types,lines kern*
```

### 3.6.6 Register Variables in Intel Symbol Tables

The -REGVARS switch includes symbolic information about register variables in an Intel style symbol table. Since the Intel formats have no definitions for register variable information, these formats have been extended to support this additional information. The extensions to the format are described in Appendix D.

The -REGVARS switch is used in conjunction with -ISYMBOLS and input modules containing CodeView symbolic information to create an Intel OMF-386 format symbol table with register variable information.

The -NOREGVARS switch disables generation of register variable information in Intel symbolic tables, so that a standard (non-extended) OMF-386 symbol table is produced. This is the default operation of 386 | LINK.

These switches take no arguments.

**Syntax:**

```
-regvars
-noregvars
```

**Short Form:**

```
-reg
-noreg
```

**Example:**

```
386LINK hello -isymbols -regvars
386LINK hello -isymbols -noreg
```

## 3.6.7 Treating Local Symbols as Data

CodeView symbol information is encoded as data records in segments named $$TYPES and $$SYMBOLS in the input modules. By default, 386 | LINK removes this data from the output executable file and processes the symbol information to create a symbol table in the selected symbol format.

The -LOCPASSTHRU switch passes any source code symbol information as standard segments, or data, through to the output file. The data segments that contain this information do not appear in the map file or the executable file. This switch allows a program to access its symbol table information at run-time as data. The CodeView typedefs reside in a segment named $$TYPES, and the local symbols reside in segment named $$SYMBOLS. The switch takes no arguments.

The -LOCDISCARD switch does not pass CodeView information through as data to the executable output file. This is the default operation of 386 | LINK.

**Syntax:**

```
-locpassthru
-locdiscard
```

**Short Form:**

```
-locp
-locd
```

**Example:**

```
386link hello -cvsym -locpassthru
386link hello -cvsym -locdiscard
```

## 3.7 386 | LINK Operation Control

The sections below describe switches that control segment ordering, checksum validation, and linker warning levels.

### 3.7.1 386 | LINK Banner Display

The -BANNER switch displays a 386 | LINK copyright message with the 386 | LINK version and serial number. This is the default operation of 386 | LINK.

The -NOBANNER switch suppresses the 386 | LINK copyright message.

The switches take no arguments.

**Syntax:**

```
-banner
-nobanner
```

**Short Form:**

```
-banner
-nobanner
```

**Example:**

```
386link hello -banner
386link hello -nobanner
```

## 3.7.2 Segment Ordering

As part of the linking process, 386 | LINK combines pieces of segments from different object modules into complete segments, then orders these segments in the executable image output file. 386 | LINK uses two different algorithms to order segments.

The default segment ordering rules, selected with -LOGORDER switch, are appropriate for most programs. The Microsoft family of 8086 real mode compilers assumes a different set of segment ordering rules, called DOS segment ordering.

The -DOSORDER switch selects the DOS segment ordering rules. It should be used when linking any program compiled with one of the Microsoft real mode compilers.

A complete description of both sets of segment ordering rules appears in Chapter 6.

**Syntax:**

```
-dosorder
-logorder
```

**Short Form:**

```
-dosorder
-logorder
```

**Example:**

```
386link hello -lib \msc\lib\slibce -dosorder
```

### 3.7.3 Checksum Validation on Input Files

Object records in input modules include a checksum file that can be used to verify correct information in the object record. The Microsoft C 5.1 compiler generates invalid checksums in object files.

The -NOCHECKSUM switch disables checking of object module checksums. This is the default operation of 386 | LINK, for compatibility with the Microsoft 5.1 tools.

The -CHECKSUM switch enables checking of object module checksums. When -CHECKSUM is specified, 386 | LINK generates an error if a bad checksum is encountered.

The switches take no arguments.

**Syntax:**

```
-checksum
-nochecksum
```

**Short Form:**

```
-checksum
-nochecksum
```

**Example:**

```
386link hello -checksum
386link hello -nochecksum
```

### 3.7.4 Warning Level Control

386 | LINK detects some conditions that can potentially cause runtime problems, but that are not necessarily errors. Warning messages for these conditions can optionally be generated on the display and in the map file. 386 | LINK supports three warning levels.

The -NOWARN switch disables all warning messages.

The -WARN switch is the default warning level of 386|LINK, and enables warning messages for the following conditions:

- There is no initial stack segment in the program.
- A 386|DOS-Extender switch was specified (see section 3.4.5), but the output file is not an .EXP file.
- Initial SS:SP, DS, or ES values were specified in an input module assembled by the Intel ASM86 assembler. These initial values are ignored by 386|LINK.
- Register variables cannot be represented in Intel OMF-386 format when -ISYMBOLS is used. Use the -REGVARS switch to enable the Phar Lap extensions to OMF-386 (see Appendix D) if your debugger supports the extensions.

The -FULLWARN switch enables additional warning messages for the following conditions:

- Overlapping data records. Overlapping data records occur in an assembly language program if the ORG directive backs up the program counter over existing code. New code overwrites the existing code. The -FULLWARN switch displays the module, segment, and location where an overlap occurred.
- Multiple initializations of common blocks with different values. The last object module processed is the one which supplies initial values to the output file.
- Pieces of a singles segment from different object modules having different segment attributes.
- CodeView typedefs that cannot be converted to Intel typedefs when -ISYMBOLS is used.
- Inconsistent segment grouping in different object modules.

**Syntax:**

```
-nowarn
-warn
-fullwarn
```

**Short Form:**

```
-nowarn
-warn
-fullwarn
```

**Example:**

```
386link hello -nowarn
386link hello -warn
386link hello -fullwarn
```

Chapter 4

# Providing Symbols to Debuggers

The sections below document how to provide symbolic information in object modules input to 386 | LINK, symbol table formats supported in output executable files, and symbol table placement in the executable file.

## 4.1 Symbol Information in Input Object Modules

386 | LINK reads symbolic information in input object modules in the following formats:

- 16-bit OMF-86 object modules can contain public symbols and segments in OMF-86 format, and local symbols, typedefs, line numbers, and other source code debug information in16-bit Microsoft CodeView format.
- 32-bit Easy OMF-386 object modules can contain public symbols and segments in Easy OMF-386 format, and local symbols, typedefs, line numbers, and other source code debug information in 32-bit Microsoft CodeView format.

The table below shows how to generate symbolic information in a format that can be processed by 386 | LINK for several popular compilers. Please see also the documentation supplied with the compiler.

**TABLE 4-1**
Compiler Switches for Generating Debug Information

| Compiler | Public Symbols | Source Debug Information |
|---|---|---|
| MetaWare High C-386, Professional Pascal-386 | automatic | -g switch, CodeView pragma |
| Zortech C++ | automatic | -g switch |
| Watcom C-386, Fortran-386 | automatic | proprietary symbol format only for Watcom wvideo debugger |
| SVS Fortran-386, C-386, Pascal-386 | automatic | proprietary symbol format only for SVS debugger, dbg |
| MicroWay Fortran-386, C-386 | automatic | not available |
| Microsoft C 6.0 (16-bit) | automatic | the /Zi switch |
| Phar Lap 386\|ASM | automatic | the -CVSYM switch |

## 4.2 Symbol Table Formats in Output Executable Files

386|LINK produces four types of symbol tables in the output executable file. Select the appropriate symbol format for the debugger you use.

- Phar Lap PubSym format is selected with the -SYMBOLS switch, and includes only public symbols (no source code debug information). Use it with assembly language symbolic debuggers, such as Phar Lap's 386|DEBUG.
- Phar Lap FullSym format is selected with the -FULLSYM switch, and includes full source code debug information. Use it with source code debuggers, such as the Phar Lap debugger 386|SRCBug, that read FullSym format symbol tables.

- Microsoft CodeView format is selected with the -CVSYMBOLS switch, and includes full source code debug information. Use it with source code debuggers, such as MetaWare's mdb, that read CodeView format symbol tables.

- Intel OMF-386 format is selected with the -ISYMBOLS switch, and includes full source code debug information. Use it with source code debuggers that read Intel format symbol tables.

The appropriate symbol table selection switch to use for several popular debuggers is shown in the table below. Please see also the documentation supplied with the debugger.

**TABLE 4-2**
Symbol Table Selection for Debuggers

| Debugger | .EXE Format | .EXP Format |
|---|---|---|
| Phar Lap 386\|SRCBug | -FULLSYM | -FULLSYM |
| MetaWare mdb | -CVSYMBOLS | -CVSYMBOLS |
| Zortech zdb | -CVSYMBOLS | -CVSYMBOLS |
| Phar Lap 386\|DEBUG | -SYMBOLS | -SYMBOLS |

## 4.3 Source Level Debugging with Popular Compilers

The following three actions are needed to produce an executable output file with a symbol table for source level debugging:

- Use appropriate compiler switches and/or pragmas to produce CodeView source debug information in the object files.

- Use the appropriate 386|LINK switch to produce the desired symbol table format in the output executable file.

- For 386 protected mode programs, the 386|LINK -ATTRIBUTES switch can optionally be used to produce a symbol table that correctly indicates which symbols are code symbols and which are data symbols. If the -ATTRIBUTES switch is not used the debugger may not always use the correct segment selector (000Ch for code symbols and 0014h for data symbols under 386|DOS-Extender) when displaying a symbol address. The correct code and data is always shown, since segments 000Ch and 0014h are aliases that point to the same memory. However, it can be confusing to see an incorrect segment selector in a debugger display; to avoid confusion use the -ATTIBUTES switch to indicate which input segments are code (er, or execute read) and which are data (rw, or read/write).

One thing to watch for: when the -ATTRIBUTES switch is used, you may get target/frame conflict errors from 386|LINK. This is the result of sloppy assembly language programming that is normally masked by the fact that 386|LINK places all code and data in a single program segment in the executable file for 386 protected mode programs. The programming error that causes target/frame conflicts is placing assembler EXTRN directives within the wrong segment block in an assembly language source file. The placement of the EXTRN directive tells the assembler, and ultimately the linker, which segment the public symbol is in. The EXTRN directive must be placed within a segment block for the segment in which the symbol is defined, or must be placed outside all segment blocks if the symbol's segment is not known at assembly time.

### 4.3.1 The MetaWare High C-386 Compiler

Use the following pragmas in the profile file for the High C compiler:

```
pragma on(CodeView);
pragma off (Postpone_callee_pops);
pragma off (Auto_reg_alloc);
```

Use the -g (debug) command line switch when you run the compiler, to generate source debug information and disable some optimizations:

```
hc386 -Hpro=pl386.pro -g -c hello.c
```

To debug with the Phar Lap 386|SRCBug debugger, link with the -FULLSYM and -ATTRIBUTES switches:

```
386link hello -lib hc386\hce -fullsym
-attr group cgroup er -attr group dgroup rw
```

To debug with the MetaWare mdb debugger, link with the -CVSYMBOLS and -ATTRIBUTES switches:

```
386link hello -lib hc386\hce -cvsym -attr group er -attr
group dgroup rw
```

### 4.3.2 The Zortech C++ Compiler

Use the -g (debug) command line switch when you run the compiler, to generate source debug information:

```
ztc -3 -c -g hello.c
```

To debug with the Phar Lap 386 | SRCBug debugger, link with the -FULLSYM and -ATTRIBUTES switches:

```
386link hello -lib zortech\plc.lib -fullsym
-attr class code er -attr group dgroup rw
```

To debug with the Zortech debugger, zdb, link with the -CVSYMBOLS and -ATTRIBUTES switches:

```
386link hello -lib zortech\plc.lib -cvsym
-attr class code er -attr group dgroup rw
```

### 4.3.3 The Watcom C-386 and FORTRAN-386 Compilers

The Watcom compilers do not currently support CodeView symbolics. Watcom provides its own proprietary source debug format and source code debugger. See the Watcom compiler documentation for details.

The Phar Lap debuggers, 386 | SRCBug and 386 | DEBUG, can be used to debug Watcom programs with public symbols only (no local symbols, source line number information or typedefs). Use the 386 | LINK -SYMBOLS switch.

### 4.3.4 The SVS C-386 and FORTRAN-386 Compilers

The SVS compilers do not currently support CodeView symbolics. SVS provides its own proprietary source debugger format and source code debugger. See the SVS compiler documentation for details.

The Phar Lap debuggers, 386|SRCBug and 386|DEBUG, can be used to debug SVS programs with public symbols only (no local symbols, source line number information or typedefs). Use the 386|LINK -SYMBOLS switch.

### 4.3.5 The Microway NDP FORTRAN and NDP C Compilers

The Microway compilers do not currently support source code debugging.

The Phar Lap debuggers, 386|SRCBug and 386|DEBUG, can be used to debug Microway programs with public symbols only (no local symbols, source line number information or typedefs). Use the 386|LINK -SYMBOLS switch.

## 4.4 Symbol Table Placement in Executable Files

The file header for a 32-bit .EXP executable file specifies the file offset and size of the symbol table. Please see Appendix D for details.

16-bit .EXE and 32-bit .REX format executable file headers do not identify the symbol table location. 386|LINK always places the symbol table immediately following the program code and data. The offset of the symbol table is obtained by calculating the file size from the information in the executable file header (please see Appendix D). The size of the symbol table is given by information contained in the symbol table header.

Phar Lap PubSym symbol table format is specified in Appendix D. Phar Lap FullSym symbol table format may be obtained by contacting Phar Lap Software.

Microsoft CodeView symbol table format may be obtained from Microsoft by ordering the C 6.0 Developer's Tools (part number 048-044-060). This

package includes a booklet called "Microsoft C Developer's Toolkit Reference," which documents CodeView symbol formats.

Intel OMF-386 symbol table format may be obtained from Intel by ordering the following publications:

- External Product Specification 386™ Object Modules Format (order number 482991)
- 386™ Object Modules Format Type Definition Records for High Level Languages (order number 482993)
- Type Checking: Guidelines for Constructing and Using Intel® Type Definition Records (order number 482994)
- 386™ Object Modules Format Type Definition Records Amendment (order number 482995)

Chapter 5

# The 386 | LINK Map File

## 5.1 Overview

This chapter describes the 386 | LINK map file. Unless 386 | LINK is instructed otherwise, it creates a map file each time a program is linked. The map file is an ASCII text file which can be printed on a line printer or viewed with a text editor. It provides the following information about the linked program:

- The command switches specified when the program was linked
- The names of the object files
- A list of the segments comprising the program
- A list of the public symbols in the program.
- An optional list of the local symbols in the program.

The primary of use of the map file is to locate functions and data variables during the debug phase of program development. The map file also provides an easy way to gather statistics about a program, such as the size of functions and the amount of data in the program.

## 5.2 Heading and Command Switches in the Map File

The first part of a map file is a heading and a list of command switches which were specified to 386 | LINK when the program was linked. The first heading line gives the version number of 386 | LINK which was used to link the program. The second line lists the target CPU for which the program was linked: 8086 or 80386. A typical heading looks like this:

```
386|LINK: 3.0 -- Copyright 1986-91 Phar Lap Software, Inc.
```

Following the heading, 386 | LINK produces a list of the object modules and 386 | LINK switches which were specified when the program was linked. This listing provides a written record of how a program is linked. It displays switches configured into 386 | LINK, switches from 386 | LINK environment variable, switches from the command line, and switches from any indirect command files.

For example, the command line, 386LINK -386 @EDITOR, results in a map file that includes a list of file names and switches both from the command line and from the indirect command file, EDITOR.CMD:

```
Command line switches:

-386 @EDITOR
main
cursor
cutpaste
fileio
scrpaint
utils
-lib \lib\clib
-symbols
-packed
```

## 5.3 Error Messages in the Map File

If errors occur during the link process, 386 | LINK displays error messages on the screen and also copies the messages into the map file immediately following the switch listing. The error messages in the map file provide a written record of their occurrence, so that they can be reviewed later.

For a program which has no stack segment and two undefined symbols, 386 | LINK writes the following error messages:

```
Warning: No stack segment.
Error: Undefined symbol "putstr" in module "HELLO" at location
00000009.
Error: Undefined symbol "_mwINIT" in module "HELLO".
```

## 5.4 Object Files in the Map File

The next section of the map file contains a listing of the object input modules that were linked together to build the executable program. The object modules are listed one per line in the order read by 386 | LINK. For each object module, 386 | LINK also displays the name of the file from which it was read. A typical input module listing looks like this:

```
Input module(s):

"HELLO" from file "hello.obj".
"PRINTF" from file "\hc386\hce.lib".
"init" from file "\hc386\hce.lib".
"FWRITE" from file "\hc386\hce.lib".
"APRINTF" from file "\hc386\hce.lib".
"CFINIT" from file "\hc386\hce.lib".
"ARG" from file "\hc386\hce.lib".
```

## 5.5 Segment Listing in the Map File

The segment listing portion of the map file contains a listing of the segments which make up the linked program. The listing contains one line for each segment of the program. For each segment, the following information is provided in the map file listing:

- The name of the segment
- The group in which the segment resides, if any.
- The segment class to which the segment belongs
- The segment type (PUBLIC, PRIVATE, STACK, or COMMON)
- The address relative to the start of the program of the first byte of data in the segment
- The size in bytes of the segment.

A typical segment listing is shown on the next page:

```
Segment map

     Name          Group          Class       Type   Offset     Size
------------  ------------  ------------  ----  --------  -------
CODE          CGROUP        CODE          PUB   00000000  0000000
HELLO         CGROUP        CODE          PVT   00000000  0000000
_MWPRINTF     CGROUP        CODE          PVT   00000010  000000E
```

For the segment type field, 386 | LINK uses these abbreviations:

| | |
|---|---|
| PVT | Private segment -- Segment is never combined with other segments; |
| PUB | Public segment -- Segment is combined with all other segments having the same name; |
| STK | Stack segment -- Segment is used for a stack at run-time; |
| COM | Common segment -- Segment is a Fortran common block. |

The offset field gives the address relative to the beginning of the program of the first byte of data in the segment. The format of the address is determined by the target CPU as follows:

| | |
|---|---|
| 8086 | Segment number + 16-bit offset |
| 80386 | 32-bit offset |

## 5.6 Public Symbol Listing in the Map File

The next section of the map file contains a listing of all public symbols used in a program and their locations. The listing contains one line per public symbol. The following information is provided about each public symbol:

- The symbol's name
- The address in memory where the symbol is located
- The object module which defines the symbol
- The segment in which the symbol resides
- The size in bytes of the symbol, if it is a communal variable.

A typical segment listing is shown on the next page:

```
Public symbols

     Name          Value       Module          Segment          Size
------------  --------  ------------  ------------  --------
errno         0000435A  errno         ?_MWSTATUS    00000004
fclose        00001724  FCLOSE        _MWFCLOSE
fflush        00001658  FCLOSE        _MWFCLOSE
fprintf       000000E0  PRINTF        _MWPRINTF
free          000024B8  CTOP1         _MWCTOP1
fwrite        00000810  FWRITE        _MWFWRITE
main          00000000  HELLO         HELLO
malloc        000024CC  CTOP1         _MWCTOP1
onexit        000024E0  ONEXIT        ONEXIT
```

The primary purpose of the public symbol listing is to get the address of symbols when a program is being debugged. This information is necessary to do such things as display a data variable or a set breakpoint in code. The exact procedure for locating a variable in memory depends on the target CPU. For the 80386, the value field in the public symbol listing gives the offset of a symbol relative to the beginning of the combined code/data segment. This value can be used directly in debugger commands. For example, suppose a variable name "count" had a value of 00001244 listed in the map file. The command, -dw 1244, can be used with a debugger to display the contents of count.

For an 8086 (or an 80386 real mode) target, the value of the public symbol consists of two parts: a segment number, plus a 16-bit offset. Under an 8086 debugger, the paragraph base address the program was loaded would have to be added to the segment number from the map file to get the paragraph base address of the segment containing a variable. This calculated paragraph base address must be specified, along with the symbol's offset within its segment, in order to reference it.

By default, the public symbol listing in the map file is sorted alphabetically by public symbol name. Using the -PUBLIST switch, the listing can also be sorted by load address. Section 3.5.6 describes the -PUBLIST switch in detail.

## 5.7 Local Symbol Listing in the Map File

When the -LOCMAP switch is used, local symbols are listed after the public symbol listing. They are listed by module and object file:

- the line number and address of the local symbol
- the data types of each symbol
- local variables
- local variables, in detail, from within functions.

The following is an illustration of the line number and address of the local symbol information:

```
Local symbols

Module "TYPES" from file "types.obj"

  #225:00000025   #226:00000025   #227:00000025   #228:00000025
  #229:00000025   #230:00000025   #231:00000025   #232:00000025
  #259:00000025   #260:00000029   #261:0000002F   #262:00000036
  #263:0000003A   #264:00000040   #268:00000047   #273:00000051
  #274:0000005D   #275:00000069   #276:00000075   #277:00000081
  #278:0000008D   #279:00000099   #280:000000A5   #284:000000B1
  #285:000000BE   #286:000000CB   #287:000000D8   #288:000000E5
  #289:000000F2   #290:000000FF   #291:0000010C   #292:00000119
  #293:00000126   #295:00000133   #296:00000140   #297:0000014D
  #298:0000015A   #299:00000167   #300:00000174   #301:00000181
  #302:0000018E   #306:0000019B   #307:000001A8   #308:000001B5
  #309:000001C2   #310:000001CF   #311:000001DC   #315:000001E9
```

Following that section is the data type of each symbol as shown below:

```
Type #0200 = Array of <char>, length = 3
Type #0201 = List
             1: <char>
             2: <short>
             3: <long>
             4: <unsigned char>
             5: <unsigned short>
             6: <unsigned long>
             7: <type #0200>
```

```
Type #0202 = List
             1: mchar @ 00000000
             2: mshort @ 00000000
             3: mlong @ 00000000
             4: muchar @ 00000001
             5: mushort @ 00000002
             6: mulong @ 00000002
             7: marray @ 00000003
Type #0203 = Structure "struct astruct_def", length = 17, types = #0201
             names = #0202
Type #0204 = Ptr to type #0203
Type #0205 = Array of <char>, length = 18
Type #0206 = Array of <ptr to char>, length = 4
Type #0207 = Array of <char>, length = 5
Type #0208 = Array of <long>, length = 20
Type #0209 = Array of <ptr to long>, length = 4
Type #020A = Array of <type #0203>, length = 51
Type #020B = Array of <type #0204>, length = 4
Type #020C = Array of <type #0208>, length = 60
```

The local variables are displayed next as shown immediately below:

```
achar <char> @ 000003D4
ashort <short> @ 000003D6
along <long> @ 000003D8
auchar <unsigned char> @ 000003DC
aushort <unsigned short> @ 000003DE
aulong <unsigned long> @ 000003E0
afloat <float> @ 000003E4
adouble <double> @ 000003E8
aint <long> @ 000003F0
auint <unsigned long> @ 000003F4
acharp <ptr to char> @ 000003F8
ashortp <ptr to short> @ 000003FC
alongp <ptr to long> @ 00000400
aucharp <ptr to unsigned char> @ 00000404
aushortp <ptr to unsigned short> @ 00000408
aulongp <ptr to unsigned long> @ 0000040C
afloatp <ptr to float> @ 00000410
adoublep <ptr to double> @ 00000414
```

The final section of local variables from within a function appears below:

```
Function main @ 0000001C  size = 00000284  <type #0219>
        sfchar <char> @ 0000054A
        sfshort <short> @ 0000054C
        sflong <long> @ 00000550
        sfuchar <unsigned char> @ 00000554
        sfulong <unsigned long> @ 00000558
        sffloat <float> @ 0000055C
        sfdouble <double> @ 00000560
        lchar <char> @ FFFFFFFF[EBP]
        lshort <short> @ FFFFFFFC[EBP]
        llong <long> @ FFFFFFF8[EBP]
        luchar <unsigned char> @ FFFFFFF7[EBP]
```

## 5.8 Listing of Undefined External Symbols

At the end of the map file, 386 | LINK provides a summary of all external symbols which were referenced in the program but never defined as public symbols. The listing looks like this:

```
Unresolved external reference(s):

sin
cos
tan
```

This listing does not state where these symbols were referenced in the programs. However, the error message listing at the top of the file does contain this information and can be used to locate references to undefined symbols.

Chapter 6

# Segment Ordering and Library Search Rules

## 6.1 Overview

386|LINK reads individual object files and object module libraries, resolving references to external symbols and writing out a single executable program file in one of the supported output file formats. 386|LINK also creates a map file containing information about the segments and public symbols in the program. At least one individual object file must be specified as input to 386|LINK, since object modules from libraries are only pulled in when an external symbol in the object module is referenced.

This chapter discusses some of the algorithms and rules 386|LINK applies during the link process.

## 6.2 Segmentation

All code and data in any 80X86 program must reside in a segment. A program, therefore, always has one or more segments. Each segment has a unique segment name and several other segment attributes. The segment attributes are assigned by the compiler (for high-level languages), or specified with the SEGMENT directive by the program author (for assembly language). The segment attributes are (1) an align type, used to specify a power of two alignment of the segment in memory; (2) a combine type, used to control the way 386|LINK combines segments; (3) a segment class (like the segment name, assigned either by the compiler or the program author), which is an ASCII string and which is used by 386|LINK when ordering segments in memory; and (4) an access type and a segment use attribute, which control segment access at run-time.

Code and/or data for a segment is typically split into several pieces, each residing in a different object module. The segment attributes for a particular segment should be the same in all object modules in which the segment occurs, with the exception of the align type attribute which can differ between modules (see Section 6.2.3). If 386 | LINK encounters conflicting segment definitions in different object modules, it will signal an error.

### 6.2.1 Combining Segments

The way segments are combined by 386 | LINK is controlled by the segment combine type attribute associated with the segment. There are two special combine types, COMMON and PRIVATE, which are discussed in the sections below. For all other segment combine types, the pieces of the segment encountered in the object modules are concatenated by 386 | LINK so that all the code and data in a particular segment is contiguous in the output file. The pieces from the various object modules are ordered according to the sequence in which they are processed during the link process. Object files are processed in the order they appear on 386 | LINK command line. Object modules in libraries are processed according to the rules specified in Section 6.3.

The example below shows how segments are combined for a simple program which has four input modules comprised of two segments: a code segment and a data segment (see Figure 6-1). The output file has the two segments ordered contiguously in memory, with the various pieces concatenated as shown in Figure 6-2.

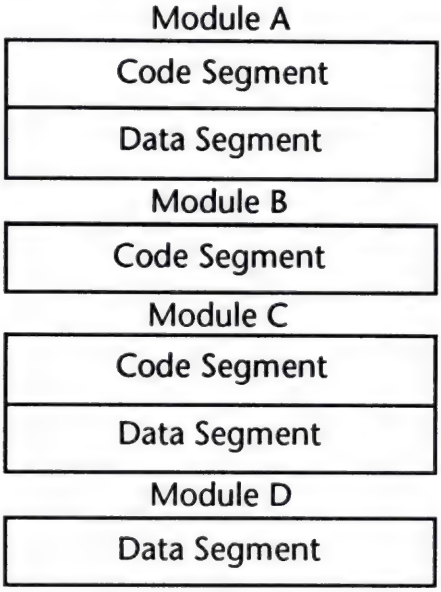



**FIGURE 6-1**
Input Modules

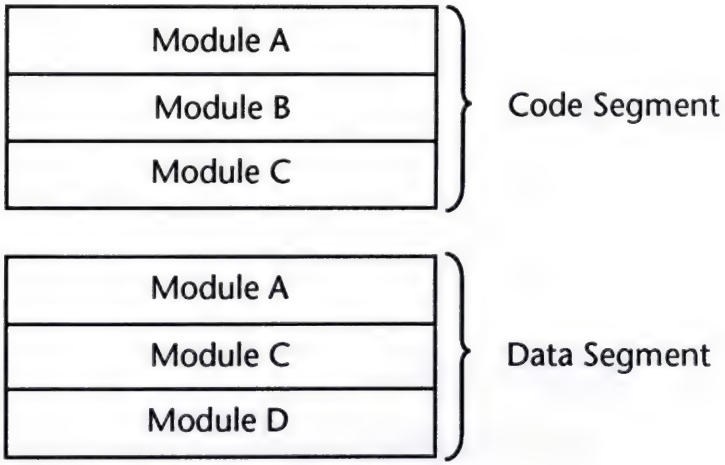



**FIGURE 6-2**
Executable Output

### Private Segments

A segment which has a PRIVATE combine type is always treated as a separate segment and is never combined with segments from other object modules, even those having the same name. Private segments are used to avoid name conflicts with segments from other object modules. They usually contain code or data which doesn't need to be referenced outside the object module in which it is defined.

### Common Segments

A segment with a COMMON align type is overlaid instead of concatenated when the different pieces of the segment are combined by 386 | LINK. A typical use of common segments is to implement common blocks in the FORTRAN programming language.

Since the segment pieces are overlaid rather than concatenated, the length of a common segment is defined as the length of the largest piece encountered by 386 | LINK. Data in a piece of a common segment (as in any segment) can either have an initial value or be specified as uninitialized. Typically, overlapping segment pieces have the same initial data values, or only one piece has initial data values and the rest have uninitialized data. If overlapping data bytes in a common segment specify different initial values, the last value encountered will be used. It is, however, risky to depend on this value because it depends on the order in which the object modules are processed. The -FULLWARN command line switch can be used to cause 386 | LINK to signal an error if conflicting data values are given in different pieces of a common segment.

## 6.2.2 Segment Ordering

Once 386 | LINK has combined all the pieces of the segments in the input files, it must order the segments in memory. Segment ordering is controlled by the order in which the segments are encountered during the link process and by the class names assigned to the segments. The following segment ordering rules are applied by 386 | LINK:

1. All segments with the same class name are ordered contiguously in memory. The class name is an ASCII string and is case-sensitive if the -TWOCASE command line switch is used.

2. Segments are ordered within a class according to the order in which they are encountered during the link process.

3. Segment classes are ordered according to the order in which they are encountered during the link process. The only exception is the null segment class. The null segment class is always placed at the end of the program, following all non-null classes.

Since segment ordering is determined largely by the order in which segments are encountered, a typical program has one module written in assembly language, which is always placed first in the link command string and which contains segment definitions for all the segments used in the program, ordered as desired for that program. Then, all subsequent object modules can use segments in any order without affecting the program's segment ordering, since the ordering has been completely specified by the first object module. For high-level languages, all object modules generated by the compiler typically have all the segments defined in the same order at the top of the object module, so that any object module can be placed first in the link command string without changing the program's segment ordering.

For example, suppose we have the three input object modules shown in Figure 6-3, containing three segments and two segment classes.

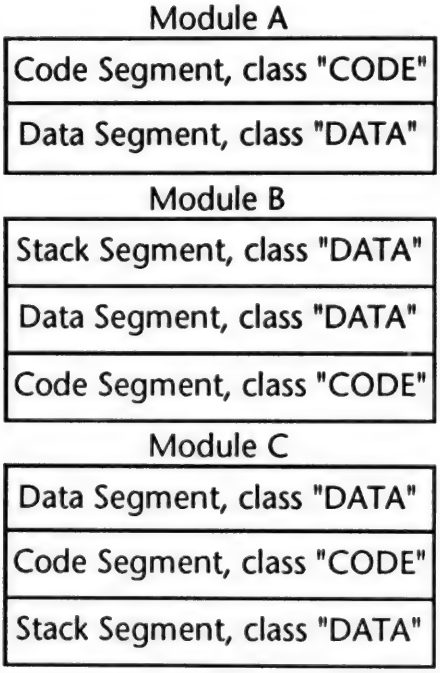



**FIGURE 6-3**
Input Modules for Segment Ordering Example

If the 386|LINK command string, 386LINK A B C, is used, then the segment ordering in the output file is as shown in Figure 6-4, with the order of the segment pieces from each input module also identified:

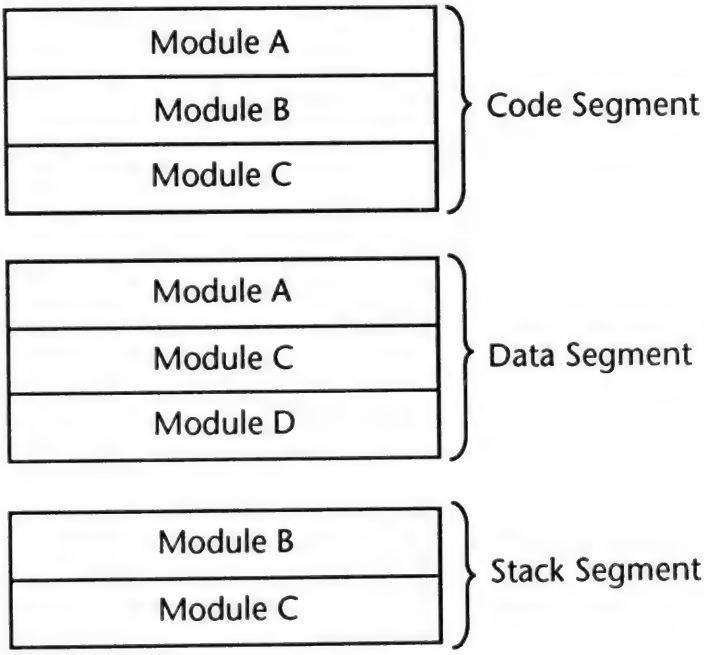



**FIGURE 6-4**
Segment Ordering Example Executable Output

Note that segments grouped with the GROUP assembler directive have absolutely no effect on segment ordering. The GROUP directive is used only to define a group of segments which can all be addressed at run-time with the same segment register. The segment ordering rules specified above must be used to ensure that all segments in a group are placed together in the output file.

The segment ordering rules described above are the rules applied by default, or when the -LOGORDER command line switch is used. The -DOSORDER command line switch is used to tell 386|LINK to use the segment ordering rules assumed by the Microsoft family of 8086 real mode compilers. The -DOSORDER switch should be used when linking a program compiled with one of the Microsoft compilers. It causes segments to be ordered according to the following rules:

1. All segments not in the group named DGROUP are placed at the beginning of the output file in the following order:

   a. All segments whose class names end with the characters "CODE";

   b. All other segments not in DGROUP, using the default segment ordering rules defined above.

2. All segments in the group named DGROUP are then placed in the output file in the following order:

   a. All segments with a class name of BEGDATA;

   b. All segments not in the classes BEGDATA, BSS, or STACK, using the default segment ordering rules defined above;

   c. All segments with a class name of BSS;

   d. All segments with a class name of STACK.

### 6.2.3 Alignment

Each segment piece in each object module has one of the following align type attributes:

| Align Type | Alignment Boundary |
|---|---|
| BYTE | Segment piece may be placed at any memory address; |
| WORD | Segment piece must be on a word boundary (address divisible by two); |
| DWORD | Segment piece must be on a double word boundary (address divisible by four); |
| PARA | Segment piece must be on a paragraph boundary (address divisible by 16); |
| PAGE | Segment piece must be on a page boundary (address divisible by 256); |
| PAGE4K | Segment piece must be on a 4-kilobyte page boundary (address divisible by 4096). |

The first piece of the first segment in the program output file is always placed at offset 0. Each subsequent segment piece is aligned as specified by the

piece's align type. Different align types may be specified for different pieces of the same segment, if desired. When aligning segment pieces, 386 | LINK pads the space between the pieces with zero bytes.

The alignment for the entire program is controlled by the program loader at the time the program is executed. The granularity of segment alignment is therefore limited by the alignment performed by the program loader. Real mode programs loaded under MS-DOS are aligned on a paragraph (16-byte) boundary. For real mode programs, the PAGE and PAGE4K align types are, therefore, not useful. 80386 protected mode programs loaded by 386 | DOS-Extender are aligned on a 4-kilobyte page boundary, so all alignment types are useful.

Normally, code segment pieces should be located on a BYTE boundary, and data and stack segment pieces should be located on a DWORD boundary, to obtain maximum throughput when fetching data. For real mode programs, the first piece of each segment is sometimes aligned on a PARA boundary to ensure that each segment is paragraph aligned at run-time. *Only* the first piece of each segment should be PARA aligned, since aligning all segment pieces on a paragraph boundary yields no advantage and wastes memory.

### 6.2.4 Communal Variables

Communal variables are a mechanism supported by the object module format for defining variables without reserving space for them in the object file. 386 | LINK automatically creates space for communal variables in the program output file. Communal variables are used in some high-level languages, such as C, to simplify the variable definition syntax. Communal variables can be defined in assembly language programs with the COMM directive.

Each communal variable has a symbol name, a data size, and a data reference type field, which indicates whether the variable is referenced as a NEAR or a FAR variable. The data reference type is determined by the memory model for which the program is compiled.

All NEAR communal variables are placed by 386 | LINK in a segment named C_COMMON. The C_COMMON segment is given a segment class of BSS and is placed in a group named DGROUP. All FAR communal variables are

placed in a segment named FAR_BSSXXXX, where XXXX is a hexadecimal number that starts at 0001 and counts up. For 8086 links, when segment FAR_BSS0001 reaches 64 kilobytes in size, additional communal variables are placed in segment FAR_BSS0002, and so on. The FAR_BSSXXXX segments are all assigned a segment class of FAR_BSS and are not placed in any group. If any segments with these hardwired names are defined in any of the object modules input to the link, any communal variables are placed following the segment data specified in the object modules.

## 6.3 Library Searching

The order in which object modules are processed by 386 | LINK is important for determining the segment ordering in memory and the order of pieces from different object modules within individual segments. Object modules which are specified as individual object files are processed in the order in which they appear on the command line. The order in which object modules pulled in from a library are processed is not always obvious, however. 386 | LINK applies the following rules when searching object libraries:

1. The libraries are searched in the order in which they appear in the 386|LINK command string;

2. Each library is searched until all possible external references, including backward references within the library, are resolved;

3. If necessary, 386|LINK recursively scans the list of libraries until all external references are resolved.

This algorithm is particularly important when two different object modules in two different libraries each have a public symbol with the same name. If both object modules are linked, because they both have public symbols which are referenced elsewhere in the program, then 386 | LINK signals a duplicate symbol error. However, if the only symbol referenced in both object modules is the duplicate symbol, then only the first module encountered is linked and no error message is generated. In this latter case, the object module which actually gets linked is determined by applying the rules listed above.

The most common reason for duplicate symbols in different modules is replacing an object module in a library for which you do not have source code available, such as a compiler run-time library. The recommended method for doing this is either to replace the object module in the library using 386 | LIB, or link the replacement routine as a single object file rather than in a library. This guarantees you will get the replacement routine rather than the object module in the library. If it is necessary to place the replacement routine in a separate library, be aware that the result (which object module actually gets linked) will be dependent on the organization and ordering of the libraries used in the link.

## 6.4 Initial Program State

The initial state of a program when it is executed is controlled by the program loader. 386 | LINK, however, must provide two pieces of information in the program file header for use by the loader: the program's entry point (its initial CS:[E]IP value), and the starting stack for the program (its initial SS:[E]SP value). See Appendix D for a specification of the program file header.

The program's entry point is specified by a record in one of the input object modules (in an assembly language program, it is specified with the END directive). If more than one object module specifies an entry point, 386 | LINK signals an error. If no entry point is specified in any object modules, no error is signaled and the initial CS:(E)IP values are set to zero. The -START switch can be used to specify an entry point at link time.

The initial program stack is placed at the end of a segment with a combine type of STACK. In all other respects, a combine type of STACK is equivalent to the PUBLIC combine type. If more than one segment in the program has the STACK combine type, 386 | LINK signals an error, unless the segment ordering rules place all segments with the STACK combine type contiguously. If there is no segment in the link with a combine type of STACK, 386 | LINK signals an error and the initial SS:(E)SP values are set to zero.

Appendix A

# 386 | LINK Switch Summary

These are the 386 | LINK command line options:

| | |
|---|---|
| 386 \| LINK *file(s)* | Input files (.OBJ) |
| -8086 | Select 8086 target CPU. |
| -80386 | Select 80386 or 80486 target CPU. |
| -80486 | Select 80386 or 80486 target CPU. |
| -ATTRIBUTE *option(s)* | Assign attributes to segment. |
| -BANNER | Display banner message. |
| -CALLBUFS *nkilobytes* | Specify the intermode call buffer size in 1K blocks for mixed real/protected mode applications. |
| -CHECKSUM | Verify checksum of object records. |
| -CVSYMBOLS | Output CodeView style symbol table. |
| -DEFINE *symbol=[seg:]value* | Define a public symbol. |
| -DOSORDER | Order segments according to the DOS segment ordering rules. |
| -DUMP | Dump object files in hexadecimal in the map file. |
| -EXE *file* | Specify the name of the .EXE or .EXP file. |
| -EXPORT *symname(s)* | Export public symbol values to the symbol file. |
| -FULLSEG | Include a full segment listing in the map file. |
| -FULLSYM | Output Phar Lap FullSym symbol table. |
| -FULLWARN | Output all warning message(s). |
| -ISTKSIZE *nkilobytes* | Specify the size of an interrupt stack in 1K blocks. |
| -ISYMBOLS | Output Intel symbol table. |
| -LIB *file(s)* | Specify library file name(s). |
| -LOCDISCARD | Discard all local symbols. |

| | |
|---|---|
| -LOCMAP | Output local symbols in map file. |
| -LOCPASSTHRU | Pass through local symbols. |
| -LOGORDER | Use logical segment ordering. |
| -MAP *file* | Specify the name of the map file. |
| -MAPNAMES *nchars* | Control length of global symbol names in the map file. |
| -MAPWIDTH *nchars* | Control the maximum line width in the program map file. |
| -MAXDATA *nbytes* | Specify the maximum number of extra data bytes to be allocated to the program when it is loaded. |
| -MAXIBUF *nkilobytes* | Specify the maximum number of 1K blocks to be allocated for the transfer buffer used for DOS system calls by 386 \| DOS-Extender. |
| -MAXREAL *nparagraphs* | Specify the maximum number of 16-byte paragraphs of real mode memory to be left free when the program when it is loaded. |
| -MINDATA *nbytes* | Specify the minimum number of extra data bytes to be allocated to the program when it is loaded. |
| -MINIBUF *nkilobytes* | Specify the minimum number of 1K blocks to be allocated for the transfer buffer used for DOS system calls by 386 \| DOS-Extender. |
| -MINREAL *nparagraphs* | Specify the minimum number of 16-byte paragraphs of real mode memory to be left free when the program is loaded. |
| -NISTACK *number* | Specify the number of interrupt stacks. |
| -NOBANNER | Suppress the banner message. |
| -NOCHECKSUM | Do not verify checksum of object records. |
| -NOLOCMAP | No local symbols in map file. |
| -NOMAP | Suppress the map file. |
| -NOOUTPUT | Suppress output executable file. |
| -NOPACK | Cancel a -pack switch. |
| -NOREGVARS | Strip register variables. |
| -NOSWITCHES | Do not list the command line switches in the map file. |

| | |
|---|---|
| -NOSYMBOLS | Do not output a symbol table with 386 \| LINK output file. |
| -NOWARN | Suppress warning messages. |
| -OFFSET *nbytes* | Specify the beginning offset of the program. |
| -ONECASE | Make symbols insensitive to case. |
| -PACK | Pack an .EXP file to save disk space. |
| -PRIVILEGED | Run program at privilege level zero. |
| -PUBLIST *option* | Specify the format in the map file of the listing of public symbols. |
| -PURGE *options* | Purge local symbols for modules. |
| -REALBREAK *symname* | Specify the number of bytes at the beginning of the program that must be loaded into real mode memory. |
| -REDEFINE *symbol=[seg:]value* | Change the value of a public symbol. |
| -REGVARS | Allow register variables in an Intel OMF-386 symbol table |
| -RELEXE *file* | Output a relocatable executable file. |
| -STACK *nbytes* | Specify the size of the stack segment. |
| -START *symname* | Specify start address of program. |
| -SWITCHES | Include a listing of the command switches in the map file. |
| -SYMBOLS | Output Phar Lap PubSym style symbol table. |
| -SYMFILE *filename* | Generate a file of exported symbol values. |
| -TWOCASE | Make symbols dual case. |
| -UNPRIVILEGED | Run program at privilege level one, two, or three. |
| -WARN | Do not suppress linker warning messages. |

Appendix B

# 386 | LINK Error Messages

## B.1 Introduction

This appendix documents the 386 | LINK error messages. Errors are grouped into five different sections based on the type of error:

- 386|LINK Fatal Errors
- 386|LINK Bad Object File Fatal Errors
- 386|LINK Errors
- 386|LINK Warnings
- 386|LINK Information Messages

If an error condition occurs that can be recovered from, 386 | LINK signals an error. The same message is also written to the map file. If an error condition occurs that prevents further processing, 386 | LINK signals a fatal error. The executable file is discarded. The map file is left in the state it was in when the fatal error occurred. It is likely that the map file is either empty or partially written.

Each of the error messages contains

- one or more probable causes
- one or more solutions, and
- in the text of the messages, themselves, often there are words in *italics*. They are a specific file name, values, offsets and the like that are substituted at the time that the error condition occurs.

Within each section, messages appear in alphabetical order. Those messages that begin with punctuation marks precede those that begin with alphabetic characters.

## B.2 386 | LINK Fatal Errors

When an error in processing occurs which makes it impossible to continue processing, 386 | LINK displays a fatal error on the screen. The .EXE file or the .EXP file is deleted, and the map file is left in whatever state it was in when the fatal error occurred. It is likely that the map file is either empty or only partially written.

This section lists all fatal errors that can occur in 386 | LINK along with the probable causes and probable solutions to the error conditions. The error messages are listed in alphabetical order for reference.

`Fatal error: Bad environment variable "variablename": setting`

***Cause:*** The 386 | LINK environment variable contains an invalid switch. An additional message displays that the problem is in the environment variable, instead of the command line or in a command file.

***Solution:*** 1) Correct the 386 | LINK environment variable.

---

`Fatal error: Bad module name`

***Cause:*** There is a syntax error in a module name that is used in a -PURGE switch.

***Solution:*** 1) Correct spelling of the module name.

---

`Fatal error: Bad number specified on command line -- "number"`

***Cause:*** A number that has been used as an argument to a command switch is invalid. The number can be invalid, because it contains a character that is not a numeric digit or because the number is too large for the switch.

***Solution:*** 1) Correct the number.

---

The following fatal errors and their causes and solutions are grouped together in the next section of this appendix. Please turn to Section B.3, 386 | LINK Bad Object File Fatal Errors for details.

```
Fatal error: Bad object file -- Bad checksum in "filename"
Fatal error: Bad object file -- Bad data record offset in
fix-up in "filename"
Fatal error: Bad object file -- Bad end record in "filename"
Fatal error: Bad object file -- Bad explicit fix-up entry in
"filename"
Fatal error: Bad object file -- Bad fix-up thread in "filename"
Fatal error: Bad object file -- Bad frame fix-up type in
"filename"
Fatal error: Bad object file -- Bad index number in module
"filename"
Fatal error: Bad object file -- Bad library file type in
"filename"
Fatal error: Bad object file -- Data record longer than 1024
bytes in "filename"
Fatal error: Bad object file -- Data record offset exceeds
segment size in "filename"
Fatal error: Bad object file -- Fix-up record has no data
record in "filename"
Fatal error: Bad object file -- Illegal group field in
"filename"
Fatal error: Bad object file -- Illegal record type in
"filename"
Fatal error: Bad object file -- Improper record length in
"filename"
Fatal error: Bad object file -- Incorrect fix-up record length
in "filename"
Fatal error: Bad object file -- Logical name defined more than
once in "filename"
Fatal error: Bad object file -- Logical name used before
defined in "filename"
Fatal error: Bad object file -- Unexpected EOF in "filename"
Fatal error: Bad object file -- Unknown file format in
"filename"
Fatal error: Bad object file -- Zero repeat count in iterative
data record
```

---

```
Fatal error: Bad symbol definition
```

***Cause:*** A syntax error has been made in a -DEFINE or -REDEFINE switch.

***Solution:*** 1) Correct the syntax error. The correct syntax is: -DEFINE name=value.

---

```
Fatal error: Can't handle repeated data blocks with a text template
```

***Cause:*** An iterative data record of an object module contains a text template that is longer than 255 bytes. 386 | LINK does not support text templates longer than 255 bytes.

***Solution:*** 1) Contact compiler vendor.

---

```
Fatal error: Cannot open command file -- "filename"
```

***Cause:*** 386 | LINK was unable to locate a command file.

***Solution:*** 1) Correct the spelling of the command file name, or create the command file.

---

```
Fatal Error: Cannot create linker output file -- reason
Fatal Error: Cannot create map file -- reason
Fatal Error: Cannot create linker virtual memory file-- reason
```

***Cause:*** 386 | LINK was unable to create one of it's output files. The error message includes the reason why it was unable to create the file. Typical problems include a full disk or a write-protected disk or directory.

***Solution:*** 1) Correct the problem. Consult the documentation supplied with the host operating system.

---

```
Fatal Error: Cannot open command file -- reason
Fatal Error: Cannot open object input file -- reason
```

***Cause:*** 386 | LINK was unable to open a command file or object input file. The error message includes the reason why it was unable to open the file. Typical problems include an incorrect file name, a nonexistent file, or a directory that is inaccessible.

***Solution:*** 1) Correct the problem. Consult the documentation supplied with the host operating system.

---

```
Fatal error: Command too long -- "command"
```

***Cause:*** In a command file, a command switch, switch value, or file name is longer than 255 characters.

***Solution:*** 1) Shorten either the command switch, the switch value or the file name that is longer than 255 characters.

---

```
Fatal error: Error accessing virtual memory file -- reason
```

***Cause:*** 386 | LINK was unable to read or write its virtual memory file. The error message includes the reason why it was unable to read or write the file. Typical problems include a full disk or a bad diskette.

***Solution:*** 1) Correct the problem. Consult the documentation supplied with the host operating system.

---

```
Fatal error: Error reading from command file -- reason
Fatal error: Error reading from object input file -- reason
```

***Cause:*** 386 | LINK was unable to read a command or object input file. The error message includes the reason why it was unable to read the file.

***Solution:*** 1) Correct the problem. Consult the documentation supplied with the host operating system.

---

```
Fatal Error: Error writing to linker output file -- reason
Fatal Error: Error writing to map file -- reason
```

***Cause:*** 386 | LINK was unable to write to the 386 | LINK output and map files. The error message includes the reason why it was unable to write the file(s). Typical problems include a full disk or a bad diskette.

***Solution:*** 1) Correct the problem. Consult the documentation supplied with the host operating system.

---

```
Fatal error: Internal system error
```

***Cause:*** An internal failure has occurred in 386 | LINK.

***Solution:*** 1) Contact Phar Lap Software, Inc., 617-661-1510.

---

```
Fatal error: Internal VM file failure
```

***Cause:*** A consistency check made by 386 | LINK has failed when accessing the virtual memory file.

***Solution:*** 1) Contact Phar Lap Software, Inc., 617-661-1510.

---

`Fatal error: Invalid callbufs value`

***Cause:*** The value given with the -CALLBUFS switch is greater than 64. The total number of 1K call buffers cannot be more than 64.

***Solution:*** 1) Reduce the -CALLBUFS value to a number less than or equal to 64.

---

`Fatal error: Invalid file name -- "filename"`

***Cause:*** A file name was given on the command line which does not follow the correct syntax for files. Example: 386LINK ...XYZ

***Solution:*** 1) Correct the spelling of the file name.

---

`Fatal error: Invalid keyword following switch "switchname" -"keyword"`

***Cause:*** A command switch, which takes a keyword as a value, has a keyword that is illegal for that switch.

***Solution:*** 1) Correct the keyword.

---

`Fatal error: Invalid nistack or istksize value`

***Cause:*** Interrupt stacks cannot be greater than 64K bytes.

***Solution:*** 1) Reduce the number of stacks by changing the value of the -NISTACK switch.
2) Reduce the size of each stack by changing the value of the -ISTKSIZE switch.

---

`Fatal error: Invalid symbol -- "symbolname"`

***Cause:*** A symbol has been specified for a command switch that is not formed properly. 386|LINK accepts symbols that follow the same rules as 386|ASM.

***Solution:*** 1) Correct the symbol. A symbol can contain only the following characters: A-Z, a-z, 0-9, _ ? $ @. The first character of a symbol cannot be a number.

---

`Fatal error: Mindata is greater than maxdata`

***Cause:*** The value specified for the -MINDATA switch is greater than the value specified for the -MAXDATA switch.

***Solution:*** 1) Correct the value of the -MINDATA switch.
2) Correct the value of the -MAXDATA switch.

---

`Fatal error: Minreal is greater than maxreal`

***Cause:*** The value specified for the -MINREAL switch is greater than the value specified for the -MAXREAL switch.

***Solution:*** 1) Correct the value of the -MINREAL switch.
2) Correct the value of the -MAXREAL switch.

---

`Fatal error: Minibuf is greater than maxibuf`

***Cause:*** The value specified for the -MINIBUF switch is greater than the value specified for the -MAXIBUF switch.

***Solution:*** 1) Correct the value of the -MINIBUF switch.
2) Correct the value of the -MAXIBUF switch.

---

```
Fatal error: Missing value to command switch
```

***Cause:*** A command switch was specified, but the value required by the switch was not given. Example: 386 | LINK hello -map

***Solution:*** 1) Specify a value for the switch.

---

```
Fatal error: No object input files were specified to be linked
```

***Cause:*** 386 | LINK was not given any object files to be linked. Example: 386 | LINK -exe myprog -map myprog.

***Solution:*** 1) Correctly specify one or more object input files.

---

```
Fatal error: Out of memory
```

***Cause:*** 386 | LINK has run out of memory for its internal tables while linking a program.

***Solution:*** 1) Remove terminate-and-stay resident utilities (TSRs) and virtual disk (VDISK) simulators to free up conventional memory below 640K for 386 | LINK.
2) Increase the amount of conventional memory on the system to 640k bytes.

---

```
Fatal error: Relocatable .EXE file cannot be generated for 8086
target
```

***Cause:*** An attempt has been made to output a relocatable executable file when the target system is an 8086 or 80286. A relocatable .EXE file can be generated only for an 80386 target.

***Solution:*** 1) Switch to another output file type, such as .EXE or .EXP.

---

```
Fatal error: Segmented relocatable .EXE file not allowed
```

***Cause:*** A relocatable executable (".REX") file can be used only with the flat memory model. This error message is generated if a multisegment link is being done and a relocatable executable file is being generated by 386 | LINK.

***Solution:*** 1) Change 386 | LINK output file to an ".EXP" file.
2) Change to the flat memory model.

---

```
Fatal error: Unexpected EOF in file "filename"
```

***Cause:*** A consistency check made by 386 | LINK has failed while accessing the object file or library named in the error message.

***Solution:*** 1) Contact compiler or assembler vendor.

---

```
Fatal error: Unexpected EOF in virtual memory file.
```

***Cause:*** A consistency check made by 386 | LINK has failed while accessing the virtual memory file.

***Solution:*** 1) Contact Phar Lap Software, Inc., 617-661-1510.

---

```
Fatal error: Unknown command switch -- "switchname"
```

***Cause:*** A command switch was given to 386 | LINK which 386 | LINK does not understand. Example: 386 | LINK hello -bad

***Solution:*** 1) Correct the command switch name.

---

## B.3 386 | LINK Bad Object File Fatal Errors

386 | LINK checks the format of all object and library files that it links. If the linker comes across an object file that contains something with an invalid format, it stops processing and displays a fatal error message on the screen, including a message explaining the problem. A bad object file error is usually caused by the assembler or compiler that generated the file.

Here are some typical reasons why an assembler or compiler might generate a bad object file:

- the original source program had errors
- the assembler or compiler has a bug
- the assembler or compiler does not follow the Intel/MicroSoft OMF-86, or the Phar Lap Easy OMF-386 standards.

```
Fatal error: Bad object file -- Bad checksum in "filename"
```

***Cause:*** The checksum byte of a record has an incorrect value, and the sum of all of the bytes of a record do not add up to zero.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product is compatible with 386 | LINK as previously noted. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad data record offset in fix-up in "filename"`

***Cause:*** The offset in a fix-up points past the end of a data record.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad end record in "filename"`

***Cause:*** An object file contains an end record with an illegal value.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad explicit fix-up entry in "filename"`

***Cause:*** An explicit fix-up contains an incorrect value.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad index number in module "filename"`

***Cause:*** An index number which refers to a segment, class, group, etc. is out of range.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad fix-up thread in "filename"`

***Cause:*** A fix-up thread references a nonexistent implicit fix-up.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad frame fix-up type in "filename"`

***Cause:*** The frame type number is not valid in a fix-up.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Bad library file type in "filename"`

***Cause:*** The library file to be linked is not a Microsoft library file or a 386 | LIB library file.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Data record longer than 1024 bytes in "filename"`

***Cause:*** A data record contains more than 1024 data bytes, which is illegal, since offsets in fix-up records cannot be larger than 1023.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

```
Fatal error: Bad object file -- Data record offset exceeds
segment size in "filename"
```

***Cause:*** A data record contains data bytes past the end of a segment.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

```
Fatal error: Bad object file -- Fix-up record has no data
record in "filename"
```

***Cause:*** A fix-up references a nonexistent data record.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Illegal group field in "filename"`

***Cause:*** A group index number is out of range.

***Cause:*** An attempt has been made to reference a logical name using an index value, and the name has not yet been defined in an LNAMES record.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link still fails with an error free file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Illegal record type in "filename"`

***Cause:*** The first byte of a record contains the type of the record. This error occurs if the type byte does not contain a valid record type number.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Improper record length in "filename"`

***Cause:*** An object record is too short to contain all the fields of the record, or the record contains extra bytes before the checksum byte.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Incorrect fix-up record length in "filename"`

***Cause:*** A fix-up record is incomplete.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Logical name defined more than once in "filename"`

***Cause:*** An attempt has been made to define a logical name more than once.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Logical name used before defined in "filename"`

***Cause:*** An attempt has been made to define a logical name more than once.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Unexpected EOF in "filename"`

***Cause:*** 386 | LINK read an object file and found no data. The object file was not complete or did not have an end record as the last record of the file.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Fatal error: Bad object file -- Unknown file format in "filename"`

***Cause:*** The object module to be linked is not an OMF-86 or Easy OMF-386 object module.

***Solution:*** 1) This error has most likely occurred because the file to be linked is not an object file.

---

`Fatal error: Bad object file -- Zero repeat count in iterative data record`

***Cause:*** The repeat count in an iterative data record is zero.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy

OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

## B.4 386 | LINK Errors

If an error occurs from which the linker is able to recover, 386 | LINK displays an error message on the screen. The identical error message is written to the map file.

This section lists all such errors that can occur in 386 | LINK. It also lists probable causes and possible solutions to these error conditions. The error messages are listed in alphabetical order for reference.

`Error: .EXE file is larger than one megabyte`

***Cause:*** The size of a program when linking for the 8086 has exceeded one megabyte in size.

***Solution:*** 1) Make the program smaller by shrinking the size of data arrays or eliminating code.
2) Investigate running the program in 80386 protected mode under Phar Lap's 386 | DOS-Extender. In protected mode, programs are not limited to one megabyte in size.

`Error: Absolute segment "segmentname" not allowed in .EXE file`

***Cause:*** An absolute segment with initialized data or code appears in a program, and an .EXE or .EXP file is being generated by 386 | LINK. Since an .EXE or .EXP file is always relocatable, the absolute segment is not allowed.

***Solution:*** 1) Remove the initialized data or code from the segment.

2) Change the type of the segment to PUBLIC in the assembler SEGMENT directive.

---

```
Error:  Attempt to put segment "segmentname" in two different groups
```

***Cause:*** In different modules, a segment is defined to be a member of different groups.

***Solution:*** 1) Change one of the group definitions so that the segment always appears in the same group.

---

```
Error:  Attempt to shrink stack size
```

***Cause:*** The value specified with the -STACK switch is smaller than the space allocated in the program. The size of the stack can only be increased, not decreased, with the -STACK switch.

***Solution:*** 1) Increase the value specified with the -STACK switch.
2) Remove the -STACK switch altogether.
3) Decrease the amount of space allocated in the program for the stack.

---

```
Error:  Because of overlapping data records, the .EXP file
cannot be packed
Warning:  Overlapping data record in module "modulename" at
offset value of segment "segmentname"
```

***Cause:*** The method used by 386 | LINK to pack an .EXP file requires that no data records overlap in a program. If 386 | LINK finds that a program has overlapping data records, it refuses to pack the file and outputs this error.

***Solution:*** 1) Modify the module so that it does not contain the overlapping data records.

---

`Error: Cannot create the segment "segmentname"`

***Cause:*** 386 | LINK creates segments for common symbols and a stack. This error is generated if the name that 386 | LINK assigns to one of these segments is already used for a public symbol or a group.

***Solution:*** 1) Rename the public symbol or group so that it does not conflict with the name that 386 | LINK needs to use for it's segments.

---

`Error: Data record missing for fixup in module "modulename" -- fixup ignored`

***Cause:*** A fixup record occurs before any data record. Since a fixup refers to a data record, this message is an indication of a bad object file.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with LinkLoc.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Error: Duplicate definition of the symbol "symbolname" in module "modulename"`

***Cause:*** An attempt has been made to define a symbol more than once.

***Solution:*** 1) Remove the extra definition of the symbol.
2) Rename one of the two symbols to a different name.

---

`Error: Fix up overflow in segment "segmentname" at offset offset Value is value`

***Cause:*** This error occurs when a value being relocated is too large to fit in the field in which the value must be stored. In most cases, this error occurs because the distance between a public symbol and the reference being fixed up is too large to fit in a 16-bit relocatable value.

***Solution:*** 1) Move the symbol definition and reference closer together or increase the width of the field in which the fixed up value will be stored.

---

`Error: Group "XXX" has overflowed.`

***Cause:*** The size of an 8086 group has exceeded 64K bytes.

***Solution:*** 1) Make the group smaller by shrinking the size of data arrays or eliminating code.
2) Move code or data to another segment.
3) Split the group into two groups.

---

`Error: Group "groupname" overlaps segment "segmentname"`

***Cause:*** All or part of group "*groupname*" is located on top of segment "*segmentname*".

***Solution:*** 1) Relocate either group "*groupname*" or segment "*segmentname*" so that they both do not try to use the same block of memory.

---

```
Error:  Multiple entry point records
```

***Cause:*** More than one module in a program has an entry point address specified.

***Solution:*** 1) Remove from the assembler END directive the duplicate entry point address.

---

```
Error:  Multiple stack segments
```

***Cause:*** More than one segment has been defined in a program with the type of STACK.

***Solution:*** 1) Remove the duplicate definition of the stack segment.
2) Give the same name to the different stack segments.
3) Change the type of one of the segments to PUBLIC on SEGMENT directive.

---

```
Error:  Program is larger than four gigabytes
```

***Cause:*** The size of a program when linking for the 80386 has exceeded four gigabytes in size.

***Solution:*** 1) Make the program smaller by shrinking the size of data arrays or by eliminating code.

---

```
Error:  Segment "c_common" has overflowed
```

***Cause:*** The segment "c_common" is created by 386 | LINK to hold common symbols allocated by 386 | LINK. This error occurs if a program is being linked for the 8086 small model, and there are more than 64K bytes of common symbols.

***Solution:*** 1) Shrink the size of the data.
2) Switch to the large 8086 model that is not limited to 64K bytes of data.

---

```
Error:  Segment "segmentname" has overflowed
```

***Cause:*** The size of an 8086 segment has exceeded 64K bytes.

***Solution:*** 1) Make the segment smaller by shrinking the size of data arrays or eliminating code.
2) Move code or data to another segment.
3) Split the segment into two segments.

---

```
Error:  Segment "segmentname" is a common segment in module
"modulename" but not in module "modulename2"
```

***Cause:*** A segment has been declared as a common segment in one module and as a public segment in another module.

***Solution:*** 1) Make the definition of the segment consistent between modules. Either make the segment always be public or always be common.
2) Rename one of the segments.

---

```
Error:  Segment "segmentname1" overlaps segment "segmentname2"
```

***Cause:*** All or part of segment "*segmentname*" is located on top of segment "*segmentname*".

***Solution:*** 1) Relocate either segment "*segmentname*" or "*segmentname*" so that they both do not try to use the same block of memory.

---

```
Error:  Target/frame conflict in end record
Error:  Target/frame conflict in segment "segmentname" at
offset offset
```

***Cause:*** When linking for the 80386, a relocatable value is used whose base is different from the segment in which the value is located. This type of reference is valid on the 8086 but impossible for the 80386.

***Solution:*** 1) Remove the segment override from the reference in the assembly language statement.

---

`Error: Thread fixup not allowed in start address`

***Cause:*** This message indicates a bad object module.

***Solution:*** 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

`Error: Undefined exported symbol "symbolname"`

***Cause:*** A symbol being exported using the -EXPORT is not defined.

***Solution:*** 1) Correct the spelling of the symbol in the -EXPORT switch.
2) Correct the spelling of the symbol in the program.
3) Make sure that the symbol is defined as a public symbol.
4) Define the symbol in the program.

---

`Error: Undefined real mode break symbol -- "symbolname"`

***Cause:*** The symbol specified with the -REALBREAK switch is not defined.

***Solution:*** 1) Correct the spelling of the symbol after the -REALBREAK switch.
2) Correct the spelling of the symbol in the source file in which it is defined.

3) Declare the symbol as public, using the assembler PUBLIC directive in the source file in which the symbol is defined.

---

```
Error:  Undefined start symbol -- "symbolname"
```

***Cause***: The symbol specified with the -START switch is not defined.

***Solution***: 1) Correct the spelling of the symbol on the -START switch or in the program.
2) Make sure that symbol is a public symbol.

---

```
Error:  Undefined symbol "symbolname" in module "modulename" at
location value
Error:  Undefined symbol "symbolname"" at location "value
Error:  Undefined symbol "symbolname" in module "modulename"
```

***Cause***: An attempt has been made to reference an external symbol that has never been defined as a public symbol.

***Solution***: 1) Define the symbol as a public symbol using the PUBLIC assembler directive.
2) Correct the spelling when it is used and when it has been declared as a public symbol.
3) Remove the external reference if it is no longer needed.

---

```
Error:  Unknown fixup type in module "modulename" in segment
"segmentname" at offset value -- ignored
```

***Cause***: An object module contains a fixup that could not be decoded by 386|LINK. This error is an indication of a bad object module.

***Solution***: 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy

OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

```
Error:  Unknown type of entry point address in module
"modulename" -- ignored
```

***Cause***: An object module contains an entry point address in an end object record that could not be decoded by 386 | LINK. This error is an indication of a bad object module.

***Solution***: 1) Rebuild the object file by recompiling or reassembling the source file. Correct all errors and re-link.
2) If the link *still* fails with an *error free* file, contact the compiler or assembler vendor to make sure that the product meets the Intel OMF-86 standard or the Phar Lap Easy OMF-386 standard. If it does not meet the standard, then it cannot be linked with 386 | LINK.
3) If the vendor does not follow the standards, contact Phar Lap Software, Inc., 617-661-1510. We will attempt to resolve any compatibility problems with the vendor.

---

## B.5 386 | LINK Warnings

This section contains the warning messages that 386 | LINK displays when it encounters a condition that is not optimum. Processing continues and the warning message prints in the map file.

```
Warning:  Cannot process all of the debug information in module
"modulename"
```

***Cause***: When translating CodeView® style debugging information to Intel style debugging information, 386 | LINK encountered a type definition that it could not decode. 386 | LINK translates the unknown type definition to null and continues

processing. If the -FULLWARN switch has been specified, 386 | LINK outputs an additional message with the number of the type definition not decoded.

***Solution***: 1) Contact Phar Lap or the compiler vendor.

---

```
Warning:  Initial SS:SP, DS or ES value ignored in module
"modulename"
```

***Cause***: The Intel ASM86 assembler allows the SS, SP, DS and ES registers to be initialized with the END directive. 386 | LINK does not support this feature.

***Solution***: 1) To initialize the SS:SP registers, make sure that the stack segment has the STACK attribute.
2) To initialize the DS and ES registers, add instructions to the start-up to explicitly set these registers.

---

```
Warning:  No stack segment
```

***Cause***: No stack was allocated for the program, either by declaring a stack segment or by allocating one with the -STACK switch.

***Solution***: 1) This warning can be ignored if the program sets up its own stack at run-time. Note, however, 386 | DOS-Extender requires a stack for any program run under it.
2) Allocate a stack by declaring a segment whose type is STACK.
3) Allocate a stack using the -STACK switch.

---

```
Warning:  Overlapping data record in module "modulename" at
offset value of segment "segmentname"
```

***Cause***: Two data records contain data bytes that overlap in 386 | LINK output file.

***Solution***: 1) Usually, this warning is caused by two different object modules initializing a common block. To correct the problem, make sure that the common block is initialized

only once. The warning is also caused by the assembler ORG directive being misused to back up the program counter, in which case, new code overwrites existing code. To correct the problem, fix the ORG statement that causes the problem not to back up the program counter.

---

```
Warning:  Putting segment "segmentname" of module "modulename"
into group "groupname"
```

***Cause***: A segment is defined to be a member of a group by one object module. A second object module also contains the same segment, but the second module does not have the segment as a member of the group. 386 | LINK automatically puts the segment in the group no matter what. If the -FULLWARN switch is turned on, 386 | LINK outputs this warning message that it is placing the segment into the group.

***Solution***: 1) Make sure that the segment is placed in the group in the module.
2) Remove the -FULLWARN switch from the link.

---

```
Warning:  Register variable(s) ignored in module "modulename"
```

***Cause***: The Intel object file formats for the 80386 do not support register variables for symbolic debugging. If register variables appear in a program and 386 | LINK is producing debugging information, then this error is output.

***Solution***: 1) While debugging a program, compile the program without register variables.
2) If the debugger understands the Phar Lap extensions to Intel OMF for register variables, then specify the -REGVARS switch when linking.

---

```
Warning:  Switch "switchname" ignored for non-EXP linker
output file
```

*Cause*: The following switches are used to specify 386 | DOS-Extender parameters: -MINREAL, -MAXREAL, -CALLBUFS, -MINIBUF, -MAXIBUF, -NISTACK, -ISTKSIZE. Since .EXP files are the only type of file that can store these 386 | DOS-Extender parameters, this error message is output if any of the above switches are specified for a 386 | LINK output file which is not a .EXP file.

*Solution*: 1) Change the type of 386 | LINK output file to an .EXP file.
2) Remove the offending switches.

---

## B.6 386 | LINK Information Messages

`Info: Virtual memory file created`

*Cause*: When 386 | LINK runs out of memory for its internal tables during a link, it begins swapping its tables to disk. This message indicates that 386 | LINK has begun swapping to disk. 386 | LINK is slower to link programs when it has to swap to disk.

*Solution*: 1) Use 386LINK, not 386LINKR.

---

Appendix C

# Object Module Formats

The format used by 386 | LINK for 32-bit object files (".OBJ" files) is called "Easy OMF-386". Easy OMF-386 is a simple extension of the OMF-86 format used by Intel and Microsoft. This appendix describes the differences between Easy OMF-386 and OMF-86. For a description of OMF-86, see references 1 or 2.

An assembler or compiler signals to 386 | LINK that an object module is targeted for the 80386 by placing the following comment record at the beginning of an object module:

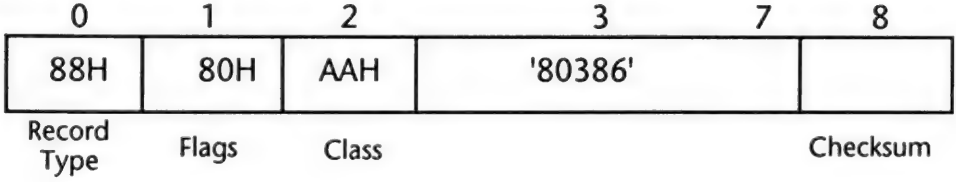



**FIGURE C-1**
Comment Record Format

The 80386 comment record should be located immediately after the module header record (THEADR) and before any other records of the object module.

The other records of an object module are formatted in Easy OMF-386 the same way as the 8086, except that any offset, displacement, or segment length field of an object record is four bytes long instead of two bytes. The following records contain fields which increase in size:

| Record: | Field: |
|---|---|
| SEGDEF | Offset and segment length |
| PUBDEF | Offset |
| LEDATA | Offset |
| LIDATA | Offset |
| Explicit FIXUPP | Target displacement |
| BLKDEF | Return address offset |
| LINNUM | Offset |
| MODEND | Target displacement |

In FIXUPP records, the following new "Loc" values have been defined:

- 5 -- 32-bit offset
- 6 -- Base + 32-bit offset (long pointer)

In SEGDEF records, the following new "align" value has been defined:

- 6 -- Segment is aligned on a 4K page boundary

An optional attribute byte is placed immediately following the overlay name index field in a SEGDEF record. The format of the attribute byte is:

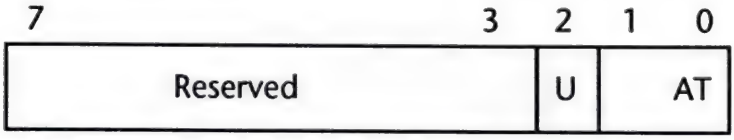



**FIGURE C-2**
Attribute Byte Format

These are the bits in the attribute byte:

- Bits 0-1 -- Segment Access Type

  00 = RO (read only)
  01 = EO (execute only)
  10 = ER (execute/read)
  11 = RW (read write)

- Bit 2 -- Segment Use Attribute

  0 = USE16
  1 = USE32

- Bits 3-7 -- Reserved, always zero

**References:**

1. Intel Corp., *8086 Relocatable Object Module Formats* (Order Number 121748-001), 1981.

2. Steven Armbrust and Ted Forgeron, ".OBJ Lessons," *PC Tech Journal*, October 1985, Vol. 3, No. 10, p. 62.

Appendix D

# MS-DOS Executable File Formats

## D.1 Introduction

The MS-DOS executable formats are for programs which will either execute in real mode directly under MS-DOS or in protected mode under control of a protected mode environment like the Phar Lap 386 | DOS-Extender. It is a binary memory image format, which also contains relocation information necessary to relocate the program at load time. There are three types of MS-DOS executable files used to target different execution environments.

- .EXE format - For real mode programs
- .EXP format - For flat model 80386 protected mode programs
- .REX format - For flat model 80386 programs which need to be relocated at load time

The type of file generated depends upon a combination of the target CPU and output format switches. Please refer to the output format switches, section 3.3, for details.

All of the MS-DOS executable file formats have the same general layout: a file header, followed by the binary data for the load image. The file header contains a signature which identifies the type of executable file, information necessary to load and relocate the file, and run-time parameters which are used by the environment under which the program will run.

## D.2 .EXE File Format

A real mode .EXE format file consists of a fixed-size header, followed by a segment relocation table, followed by the binary load image for the file. If a symbol table is selected for an .EXE file, it is appended to the end of the file (the file is larger than the size shown in the file header, and the symbol table starts at the offset given by the file size in the header). The fixed-size header is organized as follows:

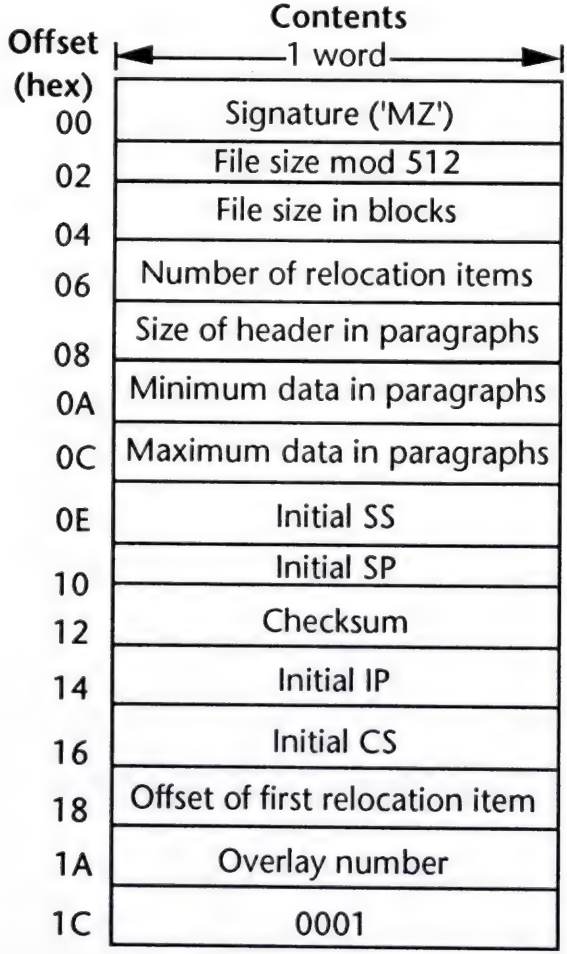



**FIGURE D-1**
.EXE File Header Format

| Offset | Contents |
|---|---|
| 00 | The file signature indicating that the file is a real mode .EXE file. The signature is the ASCII character "M" followed by an ASCII "Z." |
| 02 | The size in bytes of the file modulo 512. |
| 04 | The size of the file in 512-byte blocks. This size includes a full 512 bytes allocated to the last block. (A 513-byte file would have a value of two in this field and a value of one in the previous field.) |
| 06 | The number of items in the segment relocation table for the file. |
| 08 | The size of the file header in 16-byte paragraphs. This size includes both the fixed-size header and the variable-size segment relocation table and is used to locate the start of the program image in the file. |
| 0A | The minimum number of 16-byte paragraphs which must be allocated at the end of the program when it is loaded into memory. |
| 0C | The maximum number of 16-byte paragraphs to be allocated at the end of the program when it is loaded into memory. The hexadecimal value "FFFF" instructs DOS to allocate all available memory to the program. |
| 0E | The paragraph base of the stack in the program image. The sum of this number and the paragraph base where the program is loaded should be loaded into the SS register. |
| 10 | The initial value of the stack pointer register, SP. |
| 12 | The 16-bit checksum for the file. This value is obtained by summing all the 16-bit words in the file (modulo 64K) and then performing a two's complement negation on this sum. |
| 14 | The initial value of the instruction pointer register, IP. |
| 16 | The paragraph base of the initial code segment in the program image. The sum of this number and the paragraph base where the program is loaded should be loaded into the CS register. |

18 The byte offset from the beginning of the file to the first item in the segment relocation table.

1A Overlay number for the program. This field is always zero for the main part of a program.

1C This field always contains 0001 hexadecimal.

The segment relocation table follows the fixed-size portion of the header. Each relocation table item is a four-byte *segment:offset* pointer, identifying a 16-bit segment address in the program image which must be fixed up at load time. It is formatted as follows:

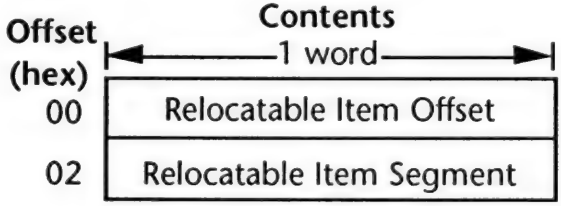



**FIGURE D-2**
.EXE File Relocation Table Entry

The first word of the pointer is the 16-bit offset, and the second word is the 16-bit segment. Once the program image is loaded into memory, the memory address of the value to be fixed up is calculated by adding the 16-bit paragraph base, where the program was loaded to the 16-bit segment address in the relocation table entry. The value in the identified location is fixed up by adding the 16-bit paragraph base where the program was loaded to the 16-bit segment value which is already in that location.

## D.3 .EXP File Format

The 32-bit protected mode .EXP format is for a flat memory model, single segment program. This is the file format used for 386|DOS-Extender applications. Segmented 32-bit file formats are supported by the Phar Lap embedded systems linker, LinkLoc.

An .EXP file is made up of several different sections. They are:

- A fixed-size file header containing general program information, along with any information necessary to locate the other sections of the program;
- A run-time parameter block which contains protected mode environment-specific switch values;
- An optional symbol table containing the program's symbol values;
- The binary image for the program's code, data, and system segments.

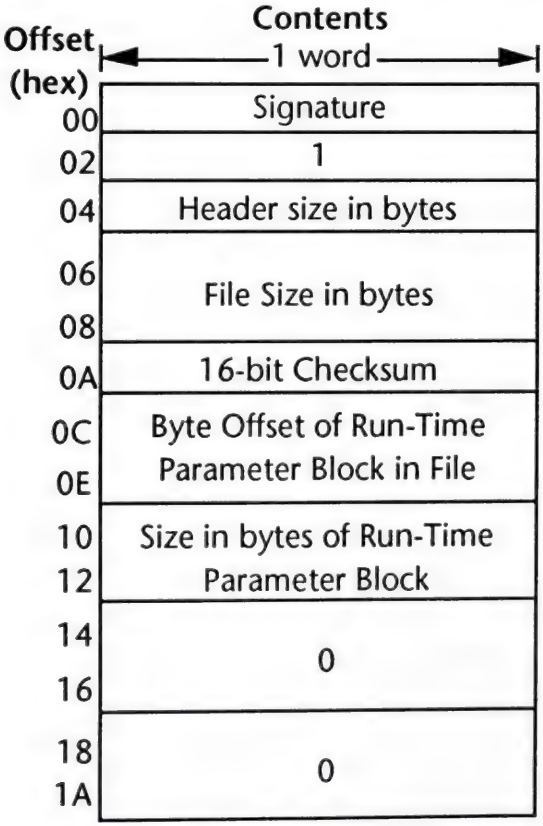



**FIGURE D-3**
.EXP File Header Format

The fixed-size portion of the header is formatted as follows:

| Offset | Contents |
|---|---|
| 00 | The file signature indicating the target CPU for the program. It is "P3" for an 80386 program. |
| 02 | The file format level. Always 0001h for a flat model program. |
| 04 | The size in bytes of the fixed-size portion of the file header. |
| 06 | The size in bytes of the entire .EXP file. This includes all header information in addition to the program's code, data and symbol table. |
| 0A | The 16-bit checksum for the file. This value is obtained by summing all the 16-bit words in the file (modulo 64K) and then performing a two's complement negation on this sum. |
| 0C | The 32-bit offset of the run-time parameter block from the start of the file. |
| 10 | The size in bytes of the run-time parameter block. |
| 14 | Always 0. |
| 18 | Always 0. |

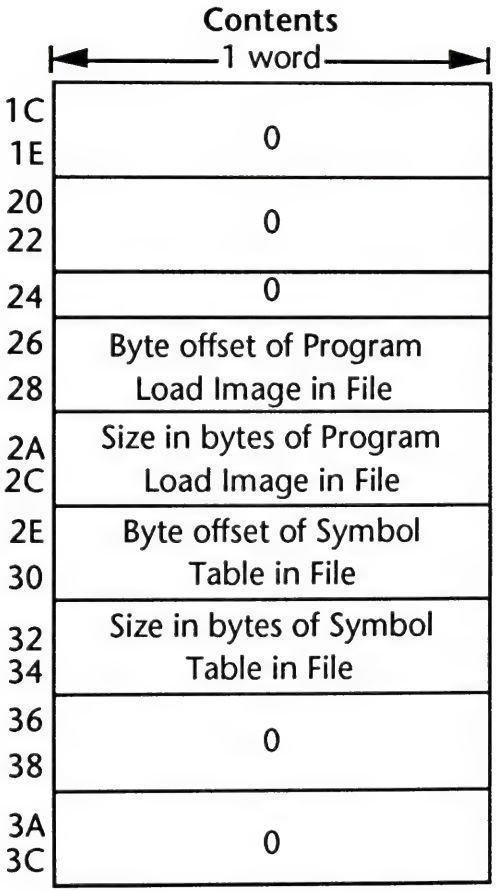



**FIGURE D-3 (cont.)**
.EXP File Header Format (cont.)

| Offset | Contents |
|---|---|
| 1C | Always 0. |
| 20 | Always 0. |
| 24 | Always 0. |
| 26 | The 32-bit offset of the program load image from the start of the file. The program load image contains all the code and data for the program. |
| 2A | The size in bytes of the program load image. |
| 2E | The 32-bit offset of the program's symbol table from the start of the file. If no symbol table is present, this field is zero. |
| 32 | The size in bytes of the symbol table. If no symbol table is present, this field is zero. |
| 36 | Always 0. |
| 3A | Always 0. |

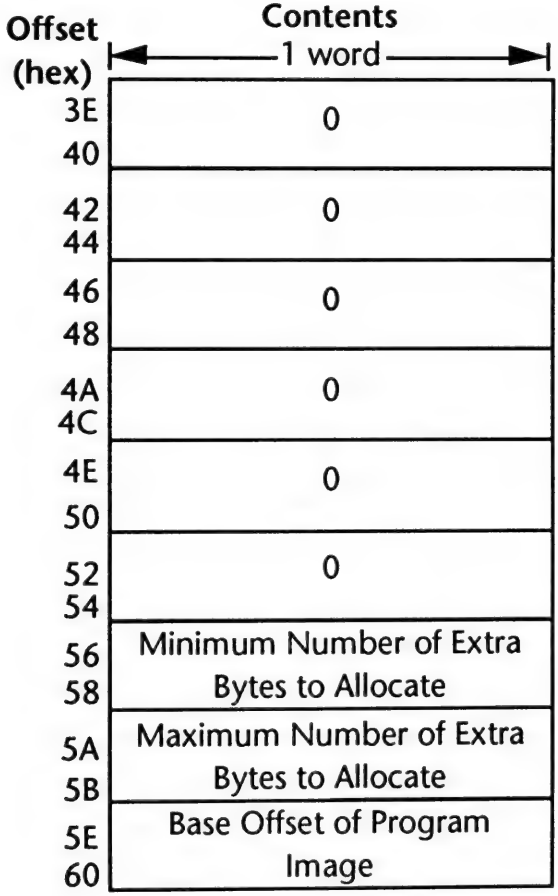



**FIGURE D-3 (cont.)**
.EXP File Header Format (cont.)

| Offset | Contents |
|---|---|
| 3E | Always 0. |
| 42 | Always 0. |
| 46 | Always 0. |
| 4A | Always 0. |
| 4E | Always 0. |
| 52 | Always 0. |
| 56 | The minimum number of bytes to allocate past the end of the program load image. |
| 5A | The maximum number of bytes to allocate past the end of the program load image. |
| 5E | The 32-bit base offset at which the flat model program segment is linked. This field is used to indicate a "hole" which appears at the head of the program load image. The run-time environment will usually use this hole to trap null pointer references. |

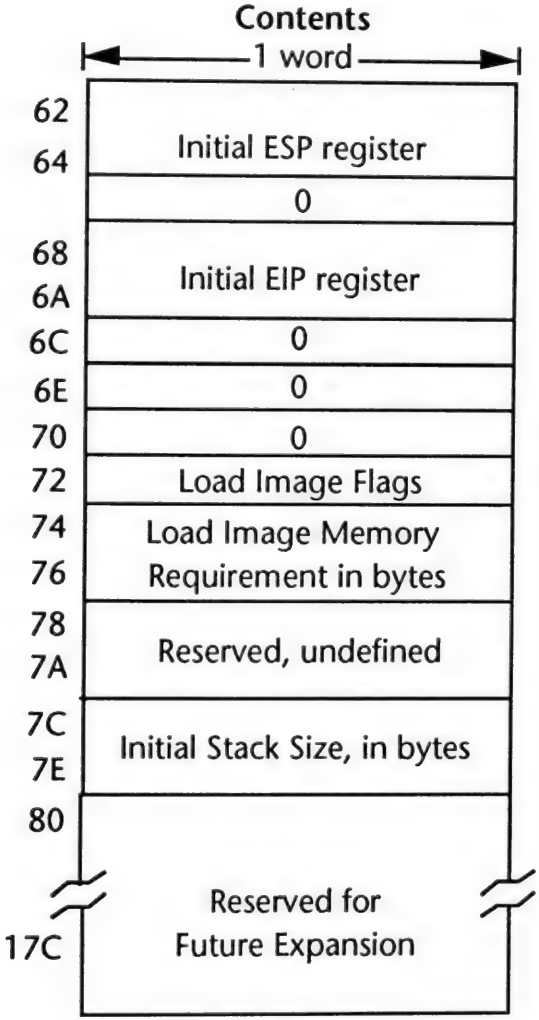



**FIGURE D-3 (cont.)**
.EXP File Header Format (cont.)

| Offset | Contents |
|---|---|
| 62 | The initial value to be loaded in the 32-bit ESP register. |
| 66 | Always 0. |
| 68 | The initial value to be loaded in the 32-bit EIP register. |
| 6C | Always 0. |
| 6E | Always 0. |
| 70 | Always 0. |
| 72 | A 16-bit flags word for the program image. The only bit currently assigned is bit zero (the low order bit), which indicates whether the program image has been compressed with the -PACK option at link time. |
| 74 | A 32-bit value which indicates the number of bytes required by the load image in the file. For an unpacked file, this value is exactly equal to the load image size. For a packed file, this value is equal to the number of bytes which the load image will require when it is unpacked. This information is necessary so that program loaders can allocate the correct amount of memory for a program before loading it. |
| 78 | Reserved, contents undefined. |
| 7C | Size, in bytes, of the initial stack. |
| 80 - 17E | These fields are reserved for future expansion and are always set to zero. |

The run-time parameter block is used to convey switch values to the protected mode environment which will load the program. For 386|DOS-Extender, this block is formatted as follows:

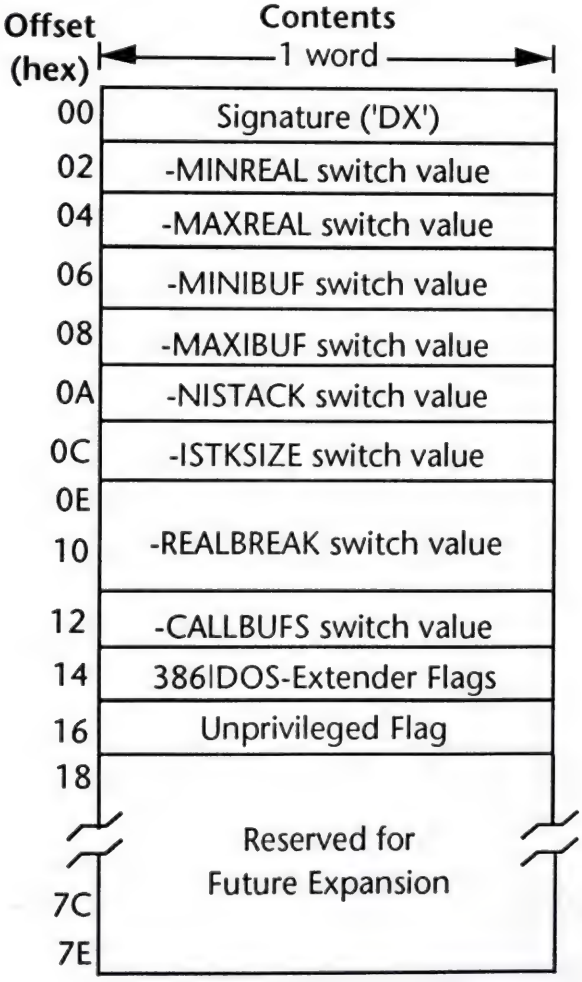



**FIGURE D-4**
Run-Time Parameter Block Format

| Offset | Contents |
|---|---|
| 00 | The signature identifying the run-time environment to which the parameters refer. This field is always "DX" for extended .EXP files created with 386\|LINK. It can be changed with a post-processor, if desired. |
| 02 | The -MINREAL switch value |
| 04 | The -MAXREAL switch value |
| 06 | The -MINIBUF switch value |
| 08 | The -MAXIBUF switch value |
| 0A | The -NISTACK switch value |
| 0C | The -ISTKSIZE switch value |
| 0E | The -REALBREAK switch value |
| 12 | The -CALLBUFS switch value |
| 14 | A 16-bit flags word. Bits 0 and 1 are reserved for Phar Lap Software, Inc. and are undefined. Bits 2-15 are currently unassigned and are always zero. |
| 16 | Unprivileged flag: set to 0001h (TRUE) if -UNPRIVILEGED switch is used, 0000h (FALSE) if -PRIVILEGED switch is used. |
| 18-7E | These fields are reserved for future expansion and are currently zero. |

Consult the *386|DOS-Extender Reference Manual* for details as to how each switch affects the operation of 386|DOS-Extender.

### D.3.1 Packed .EXP Files

.EXP files which have been compressed are identified by a bit in the load image flags word (offset 72) of the .EXP header. In a packed file, the load image portion of the file has been compressed to remove blocks of repeated data bytes. The "load image size" field of the .EXP header (offset 2A) contains the file size of the compressed load image, and the "load image memory requirement" field of the header (offset 74) indicates the size the image will have after unpacking.

The load image in a packed file is composed of a series of compressed data blocks. A compressed data block consists of a two-byte count field followed by the actual data for the block. The high bit of the count is reserved to indicate whether the block is a simple block or a repeat block. Thus, the range of values which can be specified by the count field is 0 to 32767.

If the high-order bit in the block count field is clear, then the block is a simple data block. The data for the simple block immediately follows the block count and is of the length specified by the count field.

If the high-order bit in the block count field is set, then the block is a repeat block. For a repeat block, the block count is interpreted as a byte count for the size of the memory block to be filled with the string of bytes which immediately follows it. The first byte of the string to be repeated contains the length of the string. The data bytes for the string then immediately follow the length byte. By convention, a length byte of zero is used to indicate the repeat string for a single zero byte. When expanding a repeat block, a loader should use the repeat string over and over again to fill out the block.

## D.4 .REX File Format

A .REX file is a 32-bit flat model protected mode file for a run-time environment which must relocate the program as it is loaded. If a symbol table is selected for a .REX file, it is appended to the end of the file (the file is larger than the size shown in the file header, and the symbol table starts at the offset given by the file size in the header). The .REX file header is formatted as follows:

| Offset (hex) | Contents (1 word) |
|---|---|
| 00 | Signature ('MQ') |
| 02 | File size mod 512 |
| 04 | File size in blocks |
| 06 | Number of relocation items |
| 08 | Size of header in paragraphs |
| 0A | Minimum data in 4K pages |
| 0C | Maximum data in 4K pages |
| 0E<br>10 | Initial ESP |
| 12 | Checksum |
| 14<br>16 | Initial EIP |
| 18 | Offset of first relocation item |
| 1A | Overlay number |
| 1C | 0001 |

**FIGURE D-5**
.REX File Header Format

| Offset | Contents |
|---|---|
| 00 | The file signature indicating that the file is a protected mode .REX file. The signature is the ASCII character "M" followed by an ASCII "Q". |
| 02 | The size in bytes of the file modulo 512. |
| 04 | The size of the file in 512-byte blocks. This size includes a full 512 bytes allocated to the last block. (A 513-byte file would have a value of two in this field and a value of one in the previous field. |
| 06 | The number of items in the offset relocation table for the file. |
| 08 | The size of the file header in 16-byte paragraphs. This size includes both the fixed-size header and the variable-size offset relocation table and is used to locate the start of the program image in the file. |
| 0A | The minimum number of 4K-byte pages which must be allocated at the end of the program when it is loaded into memory. |
| 0C | The maximum number of 4K-byte pages to be allocated at the end of the program when it is loaded into memory. The hex value "FFFF" means allocate all available memory to the program. |
| 0E | The initial value of the 32-bit stack pointer register, ESP. |
| 12 | The 16-bit checksum for the file. This value is obtained by summing all the 16-bit words in the file (modulo 64K) and then performing a two's complement negation on this sum. |
| 14 | The initial value of the 32-bit instruction pointer register, EIP. |
| 18 | The byte offset from the beginning of the file to the first item in the offset relocation table. |
| 1A | Overlay number for the program. This field is always zero. |
| 1C | This field always contains 0001 hexadecimal. |

The offset relocation table follows the fixed size portion of the header. Each relocation table item is a four-byte integer containing the offset in the load image of a memory location which must be fixed up at load time. It is formatted as follows:

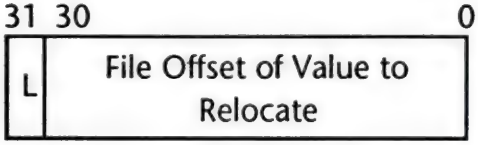



**FIGURE D-6**
.REX File Relocation Table Entry

A location is fixed up by adding the 32-bit base address for the program to the quantity which is already in the location. The high bit (bit 31) of the 32-bit offset is reserved to indicate the size of the quantity which must be fixed up. If the quantity is 16-bits wide, this bit will be clear (zero). If the quantity is 32-bits wide, this bit will be set to one.

## D.5 PUBSYM Symbol Table

A public symbol table is generated when the -SYMBOLS switch is used. PubSym format contains information about segments and public symbols, but not local symbols.

A Phar Lap PubSym symbol table consists of

- a symbol table header followed by
- a segment symbol table and
- a public symbol table.

### D.5.1 PubSym Symbol Table Header

The header consists of a signature that identifies the symbol table, and size fields for each portion of the symbol table. It is organized as shown in Figure D-7:

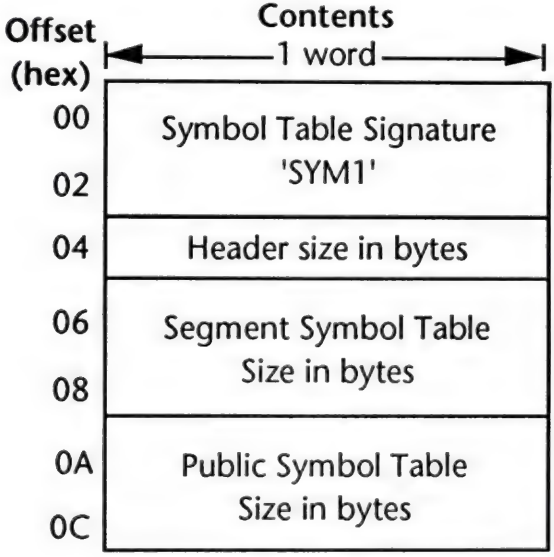



**FIGURE D-7**
PubSym Symbol Table Header Format

| Offset | Contents |
|---|---|
| 00 | Phar Lap PubSym symbol table signature. This is a four-character ASCII string which identifies the symbol table as a Phar Lap symbol table. Its value is "SYM1." |
| 04 | PubSym symbol table header size in bytes. This field can be used to locate the segment symbol table in the file. |
| 06 | Segment symbol table size in bytes. This field is a four-byte quantity which specifies the total size of all the entries in the segment symbol table. Each entry in the table is formatted as described below. |
| 0A | Public symbol table size in bytes. This field is a 4-byte quantity which specifies the total size of all the entries in the public symbol table. Each entry in the table is formatted as described below. |

### D.5.2 PubSym Segment Symbol Table

The segment symbol table immediately follows the symbol table header. It can be located in the file by adding the size of the symbol table header to the offset at which the header appears in the file. It contains one entry for each segment in the program.

For segments which reside in a group, there is a single symbol table entry for the group name and no symbol table entries for the segments making up the group. The table entries are variable length, and are packed together with no padding between them. Each entry is made up of a variable length string containing the segment name followed by a fixed size structure containing information about the segment.

Figure D-8 illustrates the format of each entry in the segment symbol table:

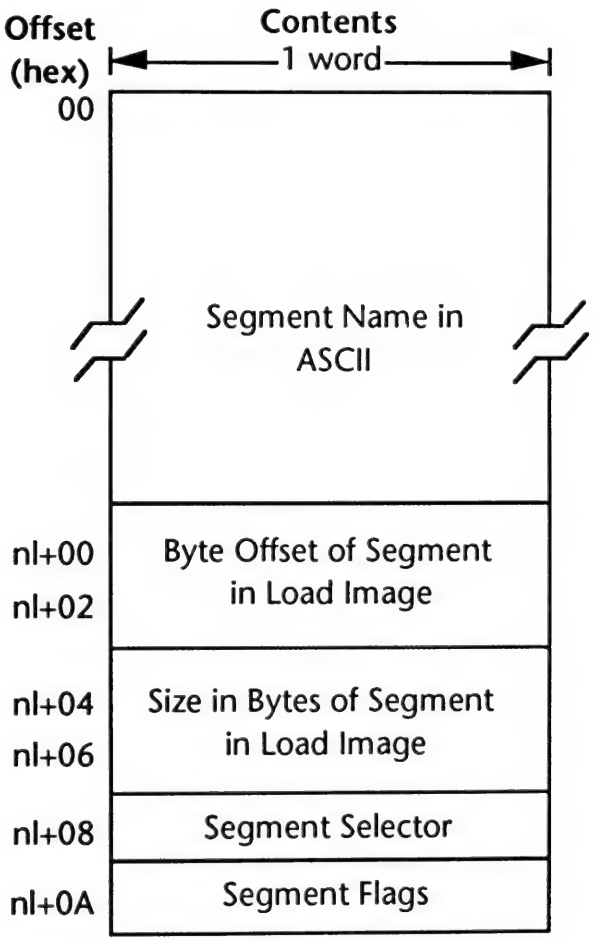



**FIGURE D-8**
PubSym Segment Symbol Entry Format

| Offset | Contents |
|---|---|
| 00 | Segment name field. The first byte of the field contains the length of the segment name in bytes. It is immediately followed by the ASCII characters for the segment name. Thus the segment name field for the segment named "SEG1" would be 5 bytes long, with the length byte containing the value four. There is no terminating character for the segment name; nor is there any padding between the last character of the name and the start of the next symbol table entry. |
| nl + 00 | nl = length of the symbol.<br>This field is a 4-byte quantity that contains the byte offset of the segment or group in the program load image. |
| nl + 04 | nl = length of the symbol.<br>Size in bytes of the segment. This is a 4-byte quantity that contains the total size of the segment in bytes. |
| nl + 08 | nl = length of the symbol.<br>This is a 2-byte quantity which contains the segment selector for the segment. For real mode programs, this is the paragraph base of the segment in the load image. For protected mode programs, this is the protected mode segment selector, which references one of the segment descriptor tables (GDT or LDT). |
| nl + 0A | nl = length of the symbol.<br>Segment flags word. The only bit currently assigned is bit zero (the low order bit) and it is used to indicate whether or not the segment is an absolute segment. If a segment in a real mode program has been located at an absolute paragraph base, this bit will be set in its symbol table entry. |

### D.5.3 PubSym Public Symbol Table

The public symbol table follows the segment symbol table and can be located by adding the size of the segment symbol table to the offset at which it appears in the file. It contains one entry for each public (global) symbol in the program. Entries are variable length, and are packed together with no padding between them. Like segment symbol table entries, each public symbol table entry is made up of a variable length string containing the symbol name followed by a fixed size structure containing the symbol's value.

Figure D-9 illustrates the format of each entry in the Public Symbol Table:

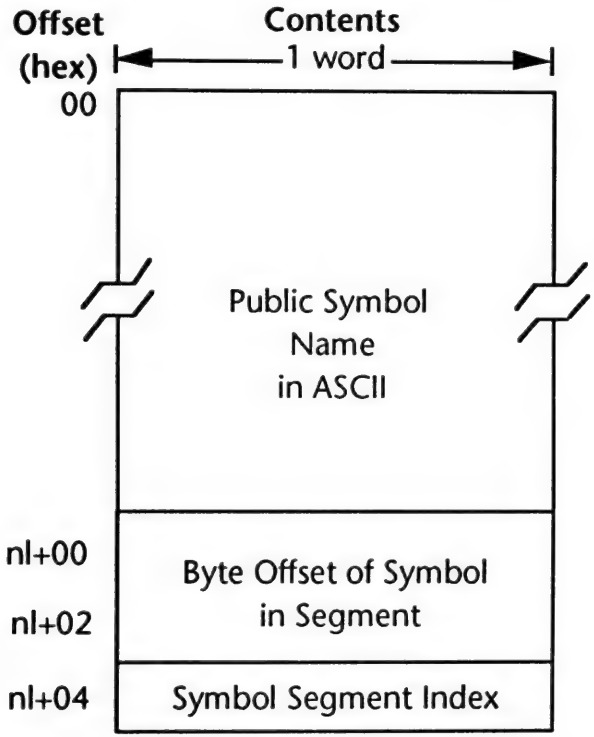



**FIGURE D-9**
PubSym Public Symbol Entry Format

| Offset | Contents |
|---|---|
| 00 | Public symbol name field. The first byte of the field contains the length of the symbol name in bytes. It is immediately followed by the ASCII characters for the name. Thus the name field for the symbol named "PUB27" would be 6 bytes long, with the length byte containing the value 5. There is no terminating character for the symbol name, nor is there any padding between the last character of the name and the start of the next symbol table entry. |

| Offset | Contents |
|---|---|
| nl + 00 | Symbol offset value. This is a 4-byte quantity that contains the symbol's offset in its segment. For symbols that do not reside in a segment (absolute constants), this field contains the symbol's constant value. |
| nl + 04 | The symbol segment index is a 2-byte field that identifies the segment in which the symbol resides. It is a 1 relative index to an entry in the segment symbol table. Thus a symbol that is in the fourth segment of the segment symbol table would have the value 4 in this field. If the symbol does not reside in a segment, this field will contain a zero. |

## D.6 Register Variables in Intel OMF-386 Symbol Tables

Storing a variable in a register is an optimization which is frequently used by the Microsoft and MetaWare compilers. The symbolic information object files generated by these compilers cannot be completely translated from CodeView to Intel symbolic formats because they cannot represent that a local variable is allocated to a register.

When 386 | LINK encounters CodeView style information that cannot be converted to Intel style information, by default, that information is discarded. This means that variables which are allocated in registers will not be visible symbolically to debuggers which support the Intel symbol formats. As a warning, 386 | LINK outputs a message that register variables were ignored. (And, if the -FULLWARN switch is used, 386 | LINK will also identify the information which has been discarded.)

To make it possible to translate symbolic information, including register variables, 386 | LINK supports an additional record for the Intel OMF-386 symbol format making full symbolic information available to debuggers. The 386 | LINK switch, -REGVARS, enables this conversion.

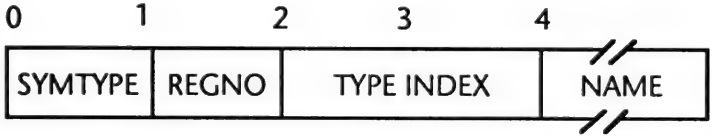



**FIGURE D-10**
OMF-386 Register Variable Record

Byte
Offset Field Name

0 SYMTYPE - Symbol record type number is hex 40. The value was chosen to avoid conflicts with the record numbers used by Intel.

1 REGNO – Valid register numbers are:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| AL | 0 | AX | 8 | EAX | 16 | ES | 24 | |
| CL | | 1 | CX | 9 | ECX | 17 | CS | 25 |
| DL | 2 | DX | 10 | EDX | 18 | SS | 26 | |
| BL | 3 | BX | 11 | EBX | 19 | DS | 27 | |
| AH | 4 | SP | 12 | ESP | 20 | FS | 28 | |
| CH | 5 | BP | 13 | EBP | 21 | GS | 29 | |
| DH | 6 | SI | 14 | ESI | 22 | | | |
| BH | 7 | DI | 15 | EDI | 23 | | | |

2 - 3 TYPEINDEX - Standard Intel OMF-386 type index that points to a typedef for the variable.

4 NAME - The name of the variable, byte 4 contains the length and bytes 5 –n contain the name

# Index



## A



## B



## C

## D



## E



## F

## G



## I



## L



## M

## N



## O



## P

## R



## S

## T



## U



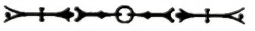
## V



## W



## Z



The people who wrote, edited, revised, reviewed, indexed, formatted, polished, and printed this manual were:

Bruce Alger, John Benfatto, Alan Convis, Noel Doherty, Lorraine Doyle, Bryant Durrell, Nan Fritz, Bob Moote, Kim Norgren, Richard Smith, Hal Wadleigh and Amy Weiss


60 Aberdeen Avenue, Cambridge, MA 02138
617-661-1510 Fax: 617-876-2972